CD-ROM include

30日で
できる!

OS自作入門

operating system

川合秀実 著

マイナビ

30日で
できる!

OS自作入門

operating system

川合秀実 著

マイナビ

●本書サポートサイト
http://book.mynavi.jp/support/pc/1984/

●本書は「30日でできる！　OS自作入門」（2006年02月紙版刊行）を元にした電子版です。
●本書の紙版にはCD-ROMが付属していましたが、電子版には付属しておりません。上記本書サポートサイトから、データをご入手ください。
●書籍中に、CD-ROMに関する記述や解説がありますが、適宜読み替えをお願いします。

●本書中の情報は、基本的に上記書籍制作段階のものです。
●本書に登場するソフトウェアのバージョン、URL、製品のスペックなどの仕様や情報は、すべてその原稿執筆時点でのものです。制作以降に変更されている可能性がありますので、ご了承ください。

・本書中に掲載している画面イメージは、特定の設定に基づいた環境にて再現される一例です。ハードウェアやソフトウェアの環境によっては、必ずしも本書通りの画面にならない場合があります。あらかじめご了承ください。

・本書に記載された内容は、情報の提供のみを目的としております。したがって、本書を用いての運用はすべてお客様自身の責任と判断において行ってください。

・本書の制作にあたっては正確な記述につとめましたが、著者や出版社のいずれも、本書の内容に関してなんらかの保証をするものではなく、内容に関するいかなる運用結果についてもいっさいの責任を負いません。あらかじめご了承ください。

・本書中の会社名や商品名は、該当する各社の商標または登録商標です。

まえがき

「OSを作ってみたい」。これはプログラマなら誰だって一度は夢見ることだと友人に言われたことがあります。「誰だって」というのはちょっと言いすぎじゃないかと思うのですが、でもプログラマの夢としてはトップテンに入るくらいのテーマかもしれません。

OSを作るというと途方もないことのように思えるかもしれませんが、それはきっとOS業界の陰謀なのです（笑）。OSは作るのがとても難しいのだと信じられていれば、自分が作ったOSに高い値段をつけて売ることができますし、OS作者は尊敬してもらえます。……では実際のところはどうかというと、他のプログラムと比べてそれほど難しくはありません。少なくとも筆者はそう感じています。

みなさんの中には、かつて一度OSの作成に挑戦したけど難しすぎてあきらめてしまった人がいるかもしれません。そんな人にとっては、筆者のこの説明には納得できないでしょう。……いえいえ違うんですよ。難しいから失敗したのではなく、簡単だよといって説明してくれる人がいなかったから失敗したのです。

これはOSの作り方に限りませんが、難しいと思っている人から説明してもらっても、やさしくて分かりやすい説明なんて期待できません。同じことを説明にするにしても、難しくややこしく説明するはずです。それじゃあ分かりにくいのは当たり前なんです。

さあ、筆者といっしょにもう一度挑戦してみませんか？　一度OSを作りたいと思ったことがあるのでしたら、きっと楽しんでもらえると思います。

■■■■■

700ページにもなる本のどこがいったい「楽しい」のか「簡単」なのかと思われるかもしれません。うーん、それを言われると弱いのですが、でもただちょっと長くなっただけなんですよ。1日分は平均22ページくらいです。ほらー、そんなに長くはないんですよ。

文章が軽いので油断するとスイスイ読み進んでしまうことがあるかもしれません。でもそれだと頭に入ってこないので、ゆっくりかみしめて読んでください。掲載のプログラムは日本語による説明と同じくらい重要なので、これもゆっくり読んでください。それさえ気をつけていれば、十分に理解できるはずです。

本書ではC言語やアセンブラを使ってOSを作りますが、でも心配はいりませんよ。これらについても本書で勉強しながらOSを作っていきます。本書のおかげでC言語のポインタがはじめて理解できた！　という人がいるくらいに、ていねいに書いています。こんなところからはじめても、30日後にはすてきなOSができあがります。どうぞおたのしみに。

▼CONTENTS

Chapter 4 四日目

C言語と画面表示の練習

Chapter 5 五日目

構造体と文字表示とGDT/IDT初期化

Chapter 6 六日目

分割コンパイルと割り込み処理

Chapter 7 七日目

FIFOとマウス制御

Chapter 8 八日目

マウス制御と32ビットモード切り替え

Chapter 9 九日目

メモリ管理

Chapter 10 十日目

重ね合わせ処理

Chapter 11 十一日目

ついにウィンドウ

Chapter 15 十五日目

マルチタスク-1

Chapter 16 十六日目

マルチタスク-2

Chapter 17 十七日目

コンソール

Chapter 18 十八日目

dirコマンド

Chapter 19 十九日目

アプリケーション

Chapter 20 二十日目

API

Chapter 21 二十一日目

OSを守ろう

Chapter 22 二十二日目

C言語でアプリケーションを作ろう

Chapter 23 二十三日目

グラフィックいろいろ

Chapter 24 二十四日目

ウィンドウ操作

Chapter 25 二十五日目

コンソールを増やそう

Chapter 26 二十六日目

ウィンドウ移動の高速化

Chapter 27 二十七日目

LDTとライブラリ

Chapter 28 二十八日目

ファイルと日本語表示

Chapter 29 二十九日目

圧縮と簡単なアプリケーション

Chapter 30 三十日目

高度なアプリケーション

Chapter 31 三十一日目

開発を終えた後で

コラム

注意：付録CD-ROMを使う前に

CD-ROMからハードディスクへコピーすると、もれなく「読み取り専用」の属性がついてしまうようです。

ファイルに読み取り専用属性がついたままだと、そのファイルを変更したり上書きしたりすることができません。これではたのしい改造ができないことになってしまいます。

読み取り専用属性をはずすには、CD-ROMからハードディスクへファイルをコピーしたあとで、そのファイルのプロパティを開いてチェックをはずしてOKを押します。

Windows2000ではフォルダのプロパティで読み取り専用属性をはずしてOKを押すと、

これらの変更をこのフォルダのみに適用するか、またはすべてのサブフォルダやファイルにも適用するか選択してください。

と表示されて、

このフォルダ、およびサブフォルダとファイルに変更を適用する

が選べます。この機能を利用して、たとえばharib00aのフォルダので読み取り専用属性をはずしてOKを押せば、中のファイルを一度にすべて読み書き可能に設定できることになります。

他のWindowsではこのような便利な機能があるかどうかは（時間がなくて）確認できませんでした。みなさんで工夫して何とかしてください。

■■■■■

tolsetに入れたQEMUは、Windows2000以降でないとうまく動いてくれないようです。

それでもっと古いバージョンのQEMUを探したところ、Windows95以降でも動くものを見つけられました。この古いバージョンのQEMUを使いたい人は、 tolset/z_tools/ のqemuフォルダの名前をqemu_ntに変更して、さらにqemu_9xフォルダの名前をqemuに変更してください。

この古いバージョンのQEMUはWindows2000以降でも問題なく動くように見えますが、一部の操作が本文での説明とは異なります。たとえばマウスをQEMUから解放するのは、Ctrl＋Altではなく、Shift＋Ctrlです。

また一部の環境では、この古いバージョンのQEMUは色がおかしくなるなどの誤動作もあるようです。それでもまったく動かないよりはマシだと思うので、必要に応じて活用してください。

■■■■■

付録CD-ROMはブータブルCDです。このCDから起動すると三十日目のOSが起動します。
以上についてよく分からないことがあれば、サポートページ（http://hrb.osask.jp/）で質問してください。

※付録CD-ROMに収録されているreadme.txtにも同じ内容が記載されています。

Chapter 0

ゼロ日目

開発を始める前に

1 はじめに

近頃は、好きなパーツを組み合わせることで、世界にたった1つの、個性的なPCを自作できるようになりました。また、適当なコンパイラ(*)を使えば、自分でゲームやツールなども作れます。ビルダーとかを使うと、簡単にホームページを自作することだってできます。そしてなんと、名著『CPUの創りかた』(*)を読めば、CPUだって自作できちゃうのです。

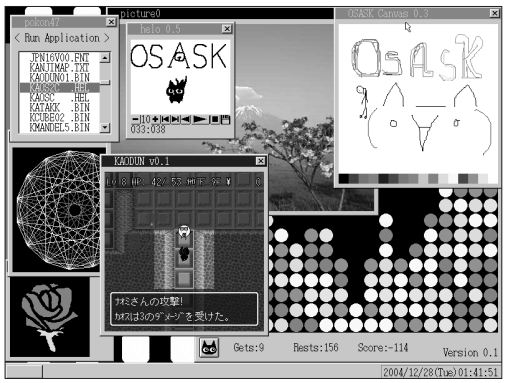
ところが、ここに手付かずで残っている分野があるではないですか。そうOS(*)の自作です。どうも初心者が気楽に挑戦できる雰囲気ではありません。PCもゲームもツールもホームページもCPUもみんな初心者が挑戦できそうな雰囲気なのに、OSだけできないなんてさみしいじゃないですか。よし、ないなら筆者が書いてしまおうということで、今回この執筆を引き受けることにしました。

たぶん初心者向きの本が少ないせいだと思うのですが、OSというと、なんだかすごく複雑で、高度なものだと思われているようです。特にWindowsやLinuxなどが、CD-ROMをいっぱいにしてしまうく

コンパイラ：ソースプログラムを機械語などへの翻訳するソフトのこと。compiler。
『CPUの創りかた』：渡波 郁 著、毎日コミュニケーションズ、ISBN4-8399-0986-5
OS：オーエス。オペレーティングシステム（operating system）のこと。WindowsとかLinuxとかMacOSとかMS-DOSとかそういうジャンルのソフトウェアの総称。

らいに大きいので、一人で趣味で作っていたら絶望的に長い時間がかかると思うかもしれません。たしかにあれだけでかいと、OSなんて一人では一生かかっても作れそうにないと筆者にも思えてきます。

しかし、心配はいりません。筆者は小さいサイズのOS開発者として実績がありまして、そのOSは80KB(*)にもならないのです。小さいとはいってもそれなりにちゃんとしたOSで、コンソール(*)しかないとか、非マルチタスク(*)だとか、そういう手抜きはないのです。



筆者が中心となって開発しているOS「OSASK」(*)

どうですか、80KBくらいなら、ちょっとがんばればできそうな気がしませんか？　初心者でもできちゃいそうな気がしませんか？　そう、1ヶ月くらいでできちゃうんです！　ということで、お気楽にいきましょう。

> コンパイル後の大きさで80KBといったら、これはプログラムとしては小さいほうだと思います。もし今まで1つでもプログラムを作ったことがあったら、そのプログラムの大きさ（.EXEファイルの大きさ）をちょっと調べてみてください。80KBというのが、どのくらいの大変さ（というか、らくちんさ）なのか想像できると思います。作ったことがない人でも、それほど難しくなさそうなフリーソフトなどをダウンロードして、実行ファイルの大きさを確認してみてください。ちなみに、Windows2000の電卓が約90KBですので、そのへんから想像してもいいかもしれません。

OSなんて作りたいと思わない・思ったこともない、という人でも、この本は面白いかもしれません。というのは、PCの自作本を読むとPCがどんな部品でできていて、どこでPCの性能が決まるのかなどが、読んでいるだけで分かります。ゲームの作り方を読むと、ゲームがどんな仕組みで動いているかが

KB：キロバイト（kilo-byte）。プログラムやデータの量を測る単位。1バイトの1,024倍。1枚のフロッピーディスクの容量が1,440KB。ちなみに1,024KBは1MB（メガバイト）。1バイトは8ビットで、ちょうど0と1の並びを8個記憶するのに必要な情報量である。Bがバイトのことかビットのことか分かりにくいということがあるかもしれないが、ここでは一部に普及した規則にのっとり、バイトは大文字Bで表し、ビットは小文字bで表すことにする。
コンソール：キーボードからコマンドを入力する仕組み。基本的にPCを文字だけで操作する。MS-DOSなどの古いOSでは主流だった操作方法。console。
マルチタスク：OSの世界では実行中のアプリケーションをタスクと呼び、複数のアプリケーションを同時に実行させることができる仕組みを、マルチタスク（multi-task）と言う。
OSASK：おさすく。筆者らが作っているOS（どさくさにまぎれて宣伝）。たったの78KBのくせに、ここまでたどり着くのに何年もの年月がかかっている。それにもかかわらず、今回作るOSが短期間でそれなりにできあがるのは、OS作成に必要な知識だけをうまくまとめたから。つまり筆者もこの本を若いときに手にしていれば、OSASKを短期間で作れていたかもしれない。そんなみなさんがうらやましい。

分かりますよね。それと同じように、OSを作る過程を通して、OSの仕組みが見えてきます。だからそういうことに関心がある人も、ぜひちょっと読んでみてください。

あとで詳しく書きますが、この本ではほとんど知識を要求しません。どんなプログラミング言語でもいいですので、簡単なプログラムなら書いたことがあるよ、くらいのセンスがあれば、それで十分です（もしかしたらそんな経験がなくても、なんとかなってしまうかもしれません）。だって初心者向きなんですからね。C言語がたくさん出てきますが、実はC言語は途中で難しくなってあきらめちゃった、という人でも大丈夫です。もちろんいろいろ知っているならそのほうが読みやすいとは思いますが、知らなくてもいいようにていねいに説明していますので、安心してくださいね。

この本では、IBM PC/AT互換機（いわゆるWindowsパソコン）を対象として、話を進めていきます。他の機種、たとえばMacintoshやPC-9821などは、多少参考になる部分もあるかもしれませんが、基本的にはそれらの機種で動かすためのOSを作ることはできません(*)。ごめんなさい。厳密にいうとAT互換機なら何でもいいというわけではなくて、386以上のCPUを搭載したものが対象です（32ビットのOSを作りたいので）。これはWindows95以降が動く機種ということになりますが、今では（中古市場を含めても）そうでないPCを探すほうがむずかしいくらいなので、きっとあなたのPCでも問題なく使えるでしょう。

メモリの容量やハードディスクの空き容量については、心配しなくて大丈夫です。ほとんどなくても、何とかなりますから。上記の条件を満たしていれば、すごく古くて動作が遅い機種でもまず大丈夫です。

2 OSってなんだろう？

正直なところを言いますと、実は筆者もよく分かりません。そんなことでこの本を書いていいのか？とおしかりを受けそうです。すみません。……いろいろなOSを見てみましたが、とても多機能なOSがあるかと思えば、ほとんど機能を持たないOSもありました。さまざまなOSを比較してみたところ、この機能が共通点、といえそうなものを見つけられませんでした。結局のところ、それぞれの作者が「これはOSなんだ」と言い張って、周囲の人も「まあそうかな」と思えばどんなソフトでもOSなんです。筆者は今のところそう思っています。

この状況を逆に利用して、筆者が最初にOSとはかくかくしかじかのものである、と自分の都合のよいことをもっともらしく言って、その条件を満たすようなソフトウェアを作ってしまう方法もありました。これも、もちろんOSを作ったことにはなると思います。たとえばMS-DOSのような、真っ黒な画面に白い文字が出てきて、コマンドを入力したら実行できる、みたいなOSを作ってもいいです。これは筆者にとってはラクです。

他の機種：この本の内容では、Macintoshだけで開発することはできないし、作ったOSをMacintoshで直接動かすこともできない。しかしPCで作ったOSを、エミュレータを使ってMacintosh上で動かすことなら多分できる。

でも、それはきっと読者のほうからすると「期待はずれ」になるでしょう。今の初心者は目が肥えていて、OSといったらWindowsやLinuxのようなものを思い浮かべるでしょう。やはりウィンドウがぽこぽこと出てきて、マウスカーソルがあって、複数のアプリケーションが同時に動くような、そんなものを期待していると思うんです。ということで、その期待にこたえるために、今回はそういうOSを作ることにします。

3 OSの作り方のいろいろ

OSの作り方にはいろいろあります。

筆者が一番いいと思っている方法は、自分の作りたいOSに近い既存のOSを見つけてきて、それを改造していくことで作ることです。これが一番短期間でできると思います。

しかしこの本では、そういうことはしないで、ぜーんぶ自分で作るということにします。というのは、読者のみなさんに、一通りのことを紹介したいと思ったからです。ベースとなるOSを探してきて、足りない機能をつけていらない機能を削る方法をこの本でやってしまうと、OSの全体の話題に触れることができません。それだと不満に思う人もいると思いました。全部作るので、話は長くなってしまいますが、まあのんびりと読んでください。筆者もじっくり書きましたので。

OSを作るとなると、たいていプログラミング言語で何を使うかという問題が出てくるのですが、今回はC言語中心でいこうと思います。あー、不満そうな声が聞こえますねえ（苦笑）。いまどきC言語なんてださいとか、C++でやってほしいよとか、Javaがいいよとか、私はDelphiが好きですとか、VisualBasicがいいとか……。気持ちは分かりますが、説明を楽にするために、ここはC言語でかんべんしてください。C言語は機能が多くなくて、でもその割には使いやすいので、ちょうどよかったんです。他の言語だと、まずその言語の説明が長くなりそうだったので、気が遠くなります。

いきなりですが、ここで1つ、一からOSを作る場合のコツを伝授します。それは、いきなり最初からOSを作ろうと思わないことです。なんとなくOSっぽく見えるものを作ればいいんです。OSを作ろうと思ってしまうと、あれもやらないといけない、これもやらないといけないなど、先々のことで頭がいっぱいになって、うんざりしてやる気がなくなります。何を隠そう、筆者自身がこれで何年も挫折を繰り返しました。ということで、この本でもOSを作るんだという意気込みは持たないで、OSっぽく見えるデモ(*)を作ることにします。そして不思議なことに、このデモを作りこんでいると、なんだかそのうちこれがデモではなくてOSになってくるのです。

デモ：デモンストレーション（demonstration）のこと。実用のためではなく、見せるための実演ソフト。

4 無知なら無知でいいじゃないか

OSを作ろうとすると、カーネルがどうだとか、シェルがどうだとか、モノリシックなのか、マイクロカーネルなのかとか、なんかそういう専門用語をまくし立てる人が、きっと横から現れるでしょう。それは時には有益なことでもあるんですが、とにかくいきなりそんなこと言われても困るのです。

そうやって横から口出ししてくる人たちを静かにさせるためには、これらについて勉強してそれなりの見解を示さないといけないのですが、はっきりいってそんなややこしいことは、入門者には不要です。勉強しているだけで時間が過ぎ、さらに世間のOSは細かいことまであれこれよく考えられているなと思い知らされて、自分はなんて浅い考えで作ろうとしていたのだろうかと打ちのめされて、やる気がなくなるだけです。もしくは先人の考えに圧倒されて、それらの技術を組み合わせるだけの作業になってしまい、面白くもなんともありません。

だからとりあえず今回は勉強なしでいきましょう。専門用語や理屈ばかり知っていても、おもしろくないです。どんなにおんぼろでも、楽しんで作れればいいじゃないですか。むしろとにかく一回作ってみて、それで問題点を把握して、その問題点を克服するためにどういう仕組みが世間では提案されているかを知るほうが、複雑な理論を深く理解できるでしょう。結果として、当面の間は口うるさい人には何も答えられないわけですが、とりあえずマイペースで自分の好きなようにしたいから、そういう話は誰か他の人とやってくださいと、お願いするしかないでしょう。

■■■■■

逆に知らないことはいいことでもあります。何も知らないので、専門家が鼻で笑っちゃうような、すごくバカらしいことも本気でやることになります。確かにたいていはそれはただのバカなのですが、たまに専門家が見落としていたすごいことを発見しちゃったりもするんです（！）。専門家はいろいろ先入観を持ってしまったせいで、試してないのにできないと思い込んでいたり、ちょっとやってやっぱりだめだと決めつけているものが結構あるんです。そんなところに挑戦できるのは私たちのような無知なしろうとだけなのです。誰でも勉強すれば専門家にはなれますが、あとになってしろうとの気持ちを取り戻すのは難しいのです。だから最初くらいは、無知なまま、できるところまでやってみようではありませんか。それで壁にぶつかったら、そのときに必要なことだけ勉強しましょう。

そもそも筆者がまさにそれを実際にやってきて、現在に至っています。プログラミングに関する学校に通ったことは一度もありませんし、難しい理論なんてほとんど勉強しないでOSを作りはじめました。しかしそのおかげで、他のOSとはかなり違った、個性的なOSになりました。専門家にもたくさんほめられましたし、今じゃ初心者向けにOSの本を書く機会までもらえているわけです。そして、作り始めてから今まで、毎日楽しんで開発してこれたのです。

そんなわけで手探りで進んでいきますので、話は分かりやすくなると思います。しかし失敗してやり直したり作り直したりもしますので、分かっている人にはもどかしいかもしれません。すみませんが、

それはガマンしてください。

なんだかこの部分を読むと勉強なんてしないほうがいい、みたいに読めますが、そんなことはありません。仕事でプログラムを作らなきゃいけないとか、一年間でこれだけやらないといけないと決まっている場合とかは、わざわざ遠回りしているヒマはありませんので、余計な失敗をしないために勉強してから取りかかるほうがずっといいです。でも今回は趣味で楽しんで作るんです。趣味くらいのんびりマイペースでやりたいじゃないですか。だからこれでいいのです。

5 OSを作るにはどうしたらいいの？

OSといえば、普通は電源を入れたら自動的に始まるものです。そんなものはどうやったら作れるのでしょうか。普通にWindowsで作った実行ファイル（〜.exe）は、OSが起動したあとにダブルクリックしたりしないと始まってくれません。私たちが今回作りたいと思っているのはそんなものではなくて、OSの入ったCD-ROMやフロッピーディスクをPCに入れておくか、もしくはハードディスクにOSを入れておけば、あとは電源を入れれば自動的に始まるような、そんなものです。

そういうものを作るために、今回は次の手順で作ります。

Windowsなどを使って**ソースプログラム**(*)を書く

↓

それをCコンパイラでコンパイルして、**機械語**(*)がいっぱい詰まったファイルを作る

↓

そのファイルを加工してフロッピーディスクのイメージファイルを作る

↓

イメージファイルをディスクへ書き込んで、OSがインストールされたディスクを作る

つまりOSを作るということは、とにかくなんとかして「自動で起動するOSが入ったディスク」を作ることなのです。

ここで出てきたイメージファイルというのは、簡単に言うと、フロッピーディスクのバックアップデータです。私たちはある内容のディスクを作りたいのですが、ディスクに磁石を当てて、望みのデータをうまく並べることはちょっとできません。だからまずバックアップデータをでっち上げて、そのデータをディスクへ書き込んでやることで、目的のディスクを作ろうというわけです。

ソースプログラム：機械語などを生成させるために書く、プログラムのこと。コンパイラで翻訳すると目的の機械語になる。source program。
機械語：CPUが直接理解できるプログラム。当然0と1だけで書かれている。でも、実はソースプログラムだって0と1だけで書かれているんだけどね（後述）。

フロッピーディスクは総容量が1,440KBなので、バックアップデータであるイメージファイルもぴったり1,440KBのファイルです。ディスクイメージを自由に作れるようになると、どんな内容のディスクでも自由に思い通りに作れるようになります。

ここでちょっと注目してほしいのですが、OSを作るためにWindowsという別のOSを利用しています。これはどうしてかというと、テキストエディタやCコンパイラを使うには、とりあえずなにかOSがないと困るからです。それでは、世界初のOSはどうしたのでしょうか。世界初のOSを作るために利用できるOSなんて、もちろんありません。だから人間がCPUの命令コード表を見て、0と1を自分で並べて、それをディスクに書き込んだのです（たぶん当時はディスクではなく、また別の記憶装置だったろうと思いますが）。これはとても大変な作業です。だからおそらく最初のOSは、きっとすごく機能を減らしたやつで、そのOSができたらそのOSを使って少しまともなOSを作って、その次はそのOSで実用的なOSを作って……みたいにしたと思います。

今回は初心者の多くがWindowsユーザだろうと思ったので、Windowsを使うことにしました。バージョンはWindows95/98/Me/2000/XPの、どれでもOKです。Linuxがいいよーという人もきっといると思います。それもそうだなと思うので、Linuxでのやり方はサポートページ(*)にまとめておきます。必要な人はぜひ見てください。

Cコンパイラやイメージファイル作成ツールなどについてですが、これが違うと細かい違いが生じて説明が大変になるので、全部付録のCD-ROMに入れてしまいました。ほとんどは筆者がリリースしているフリーソフトで、これらは先のOSASKの開発のために必要になって筆者が作ったものです。このツール類のソースプログラムも公開されています。これらのツールの他にもいくつかのフリーソフトを使いますが、それらも含めて、使うときになったら、使い方などを詳しく紹介します。

6 OSを作るときの苦労

世間で出回っているCコンパイラは、WindowsやLinuxのアプリケーションなどを作ることを前提に設計されていまして、それ以外のもの、たとえば独自のOSを作ることなんてほとんど考えられていません。筆者が提供しているCコンパイラも、結局はWindows用のgcc(*)というCコンパイラをちょっと改造しただけのものなので、状況はほとんど変わりません。OS作り対応のCコンパイラというものがもしかしたらあるのかもしれませんが、筆者にはちょっと分かりません。仮にあっても、OS開発企業くらいしか買わないと思うので、おそらく高価なものでしょう。今回はそういうものは使えません。

ということで、アプリケーション開発用のCコンパイラなどを使って、どうにかしてOSを作るしかないのです。つまり無理しているわけです。無理している以上、いろいろ不便なことがあります。

サポートページ：http://hrb.osask.jp/
gcc：ジーシーシー。GNUというプロジェクトが作ったフリーソフトのCコンパイラ。「GNU C Compiler」の略。同じスペル＆同じ発音で、GNUが作った各種コンパイラのセットを指すこともある（GNU Compiler Collection）。

たとえば printf("hello¥n"); なんてC言語の教科書の最初に出てくるのですが、これがもうできません。どうしてかというと、printfという機能は、OSが提供してくれる機能を前提にして作られているからです。私たちは、最初は何も提供できていません。だからいきなりこれを無理矢理実行させると、CPUは一般保護例外(*)を起こして、ストライキに入ってしまいます。使えない関数はprintfだけではありません。ほとんどの関数が使えません。

> いいわけくさいことを言わせていただくと、今回C言語を選んだのは、C言語にはそういうOSの支援を前提にした機能が、まあまあ少なかったからです。基本的には関数が使えないだけですみますので。これがC++になると、new/deleteなどの基本的で大事な演算子が使えなくなるとか、クラスの作り方にもいろいろ注文が入るとか、C++のいいところが生かせません。もちろん、これらを使えるようにOSを作っていけば、いずれは克服されていきます。しかし、そこまでの苦労が、C++でOSを作っているのか、それともC++にOSを作らされているのか、なんかそう思ってしまうとむなしいわけです。他の言語だと状況はもっとつらくなりそうでしたので、今回はC言語にしました。ということで、許してください。

ちなみにOSを作るときになっても一切制限がない言語は、たぶんアセンブラ(*)だけです。アセンブラ最強！（笑）(*)。でも、アセンブラだけでOSを作りましょう、なんていう本になったら、ほとんど読んでもらえないのは目に見えていたので、向こう見ずな筆者でもさすがにそれはできませんでした。

> またOSを作るときには、CPUのOS制御用のレジスタ(*)をいろいろいじる必要があるのですが、普通のCコンパイラはアプリケーションを作るためのものなので、OS制御用のレジスタを操作する命令なんて1つもありません。また、Cコンパイラは気の利いた最適化をしてくれますが、かえってそれがあだになってしまうことすらあります。

で、これらの問題を克服するために、C言語で書けない部分はアセンブラで書くしかありません。そのときに、Cコンパイラがプログラムをどういうふうに機械語に翻訳しているかを意識しないといけません。それに合わせてあげないと、C言語で作った部分とうまく情報をやり取りできないからです。こんなことは普通にC言語でプログラムしていたら味わえませんよ！　でも優越感よりも、めんどうさのほうが強いですけどね（苦笑）。

> 同様にC++でOSを作るつもりなら、C++がどのように機械語に翻訳されているかを知らなければいけないのです。当然のことながら、C++はC言語よりも多機能ですので、翻訳の規則はよりいっそう複雑で説明がめんどうなので、ここにも今回はC言語でいくことにした理由があるわけです。結局、自分が使っている言語がどういう機械語に翻訳されているのか多少は理解していないと、その言語でOSを作るのはまず無理なのです。

一般保護例外：PCのCPUは非常に優秀で、OSの保護を無視しようとしたり、ありえない動作を指示されたりすると、とりあえずその状態を保存して実行を中断し、あらかじめ設定しておいた関数を呼び出してくれる機能がある。この仕組みを例外と言っていて、除算例外とか無効命令例外とかスタック例外とかいろいろある。その中でどれにも分類されていないタイプの異常事態が一般保護例外である。古くからのWindowsユーザにとっては、この例外は青画面という悪夢を思い出すかもしれないが、OS開発を経験すると、この例外の仕組みが非常にありがたく思えてくる（後述）。
アセンブラ：機械語に一番近いと言われるプログラミング言語。かつてはこれを理解していると尊敬されたりしたのだけれど、今は変態扱いされるほうが多いかもしれない。悲しいね。本来はアセンブリ言語という言い方のほうがおそらく正式で、アセンブラは翻訳ソフトのことを指していると思われるが、筆者のような古くからのプログラマは、両者を区別しないでアセンブラという人が多い。assembler。ちなみに、本文中では一般的な記法を選択して「アセンブラ」とか「コンパイラ」と書いているが、これを読むときはもとの英語に敬意を示して「アセンブラー」「コンパイラー」などと語尾を延ばして読んでほしい。コンピュータ用語では最後が「ー」で終わる外来語が多いので、短い単語を除いて、これを省略して書くことになっている。
アセンブラ最強：ここまでではどうして最強なのかいまいち分からないと思うが、先を読むとしみじみと感じるようになるだろう。
レジスタ：機械語における変数のようなもの。CPUにとってメモリは「外部記憶装置」であって、CPUのコアに内蔵されている記憶回路はレジスタしかない。レジスタを全部合わせても、たいていは1KBにもならない。register。

世間に出回っているC言語の本やC++の本、もちろんDelphiもJavaもその他の言語の本も、みんな「結局どういう機械語に翻訳されているのか」という話をめったに書いてくれません。それどころかプログラムを通じてCPUに命令を出しているのに、そのCPUの基本的な仕組みだって説明してくれません。OS屋(*)としてはとてもさみしいです。しょうがないので、この本ではそのへんから話を始めます（今回のOS開発に必要な最低限度の範囲だけですけどね）。

こういう経験をすると、プログラミングに対する考え方が変わってくるかもしれません。今まではきれいでかっこいいソースプログラムを作ろう、とだけ思っていたのに、どんな機械語に翻訳されるかが大事なこともあるんだ！　と気がつくわけです。ソースがどんなにかっこよくても、自分の期待する機械語になってくれないなら、うまく動かないので意味がないのです。逆にソースが汚くても、さらには特定のCコンパイラでしかコンパイルできなくなったとしても、とにかく目的の機械語になってくれればそれでいいわけです。目的の機械語さえ得られれば、そのあとはソースプログラムを捨てちゃってもいいんだ！　とまでは言いませんが、でもそれくらいの気持ちでもいいくらいなんです。

OS屋にとっては、ソースプログラムなんて、結局は機械語を得るための「手段」であって目的ではないのです。手段にこだわりすぎて、かえって手間になったら本末転倒なのです。

ああところで、心配する人がいるかもしれないので言っておきますが、OSをC言語とアセンブラで作ったからといって、そのOSではC++で作ったアプリケーションは動かなくなるとか、そういうことはありません。アプリを作るときの言語は、OSがどんな言語で作られたかということとは関係ないのです。だからその点は安心してくださいね。

7　1章以降を読むにあたって（重要！）

1章からは実際の開発の日々が書かれています。1日ずつに分かれていますが、これは筆者の現在の能力と説明の長さで適当に区切ったものですので、読者のみなさんが1日分を1日でやらなければいけないということはありません。人によってどこが難しいと思うかはさまざまなので、1日分で1週間かかることもあれば、3日分を1日で理解してしまうことだってあるでしょう。

当然のことながら、読んでいるとよく分からないところがあるでしょう。そんなときは、とりあえずそのまま1〜2日分は読み進めてみるといいかもしれません。そうすれば急に分かるようになることがあります。それでも分からないときもあると思います。そういう時は、ではどこまでならきちんと理解していたかを、確認してください。そしてそこまで戻ればいいのです。あせっちゃいけません。戻って出直してみると、スラスラと分かってくることがあります。

OS屋：おーえすや。魚を売っている人は「魚屋さん」、おかしを売っている人は「おかし屋さん」だが、ここでいう「OS屋さん」というのはOSを売っているわけではない（もちろんOSを売る人もOS屋さんだ）。OSを作ったり研究したりする人。数学の研究者を「数学屋さん」、化学の研究者を「化学屋さん」と呼ぶことがあるが、それに近い。

どこまでが分かって、どこからが分からないのか、それがはっきりしていて、さらに分からないところを何度読み返しても、どうしても理解できない、ということもあるかもしれません。そういうときは、サポートページ(*)を確認してください。Q&Aページに解説があるかもしれません。

■■■■■

この本ではC言語のポインタや構造体の説明の仕方が、他の本とはだいぶ違います。それはこの本ではまずCPUの基本的な仕組みを知って、その上でアセンブラをやり、そのあとにC言語を勉強することになるからです。他の本では、そういう基礎的な話はあまりしないまま、話がポインタになった瞬間に、いきなり変数のアドレスがどうだとか言います。だから、もし他の本でポインタを分かった「つもり」になっているのでしたら、この本でのポインタの説明を読み飛ばさないでください。きっと分かるようになります。本当によく分かっている人は、適当に流し読みしていいです、もちろん。

■■■■■

これから少しずつOSを組み立てていきますが、少し進むたびにその途中段階をまとめています。それは付録のCD-ROMにまとめられていて、それをコピーすればすぐに実行させてみることができるようになっています。このプログラムに関しての注意点があるので、それを書きたいと思います。

たとえば最初に出てくるプログラムは「helloos0」というもので、その次に出てくるのは「helloos1」なのですが、helloos0に対して、本文で説明されている作業を全部やれば、それだけでhelloos1になるのかというと、そうではありません。それで十分な場合もたまにはありますが、たいていは明確には説明されていないこともやっています。それはたいてい、もういちいち細かく説明しなくてもきっと分かるだろうと思われたから、説明してないだけです。

で、結局何が言いたいのかというと、本文を読むだけではなくて、ちゃんとプログラムを見てください。本文の説明があいまいで分かりにくかったけど、プログラムを見たら一発で分かった、なんてこともありえます。この本の主役は本文ではなくて、付録のプログラムなのです。付録のプログラムが、どうやってできていったのか、その紹介を本文でやっているだけなのです。

紹介は紹介であって、完全な説明じゃないのです。くれぐれも間違えないようにしてください。……そういう意味では、「本と付録のCD-ROM」ではなくて、「CD-ROMと付録のぶ厚い本」なのかもしれません（笑）。

■■■■■

さてこのプログラムに関してもう1つ。ここに収められたプログラムの著作権は、すべて筆者にあります。しかし、この本を読んでOSを作ってみようと思ったら、いろいろまねしたい部分があるでしょう。

サポートページ：http://hrb.osask.jp/

もしかしたら初期のややこしい部分を丸ごと使いたいと思うかもしれません。さらにこの本の一番最後の状態から、まるで続きを作るかのように、OS開発を始めてみたいと思う人もいるかもしれません。

これは教材OSなのですから、みなさんがそう思ったときに、自由に使えないと意味がないと筆者は考えました。したがって自由に使っていただいてかまいません。事前に申告などはしなくていいです。そのOSには結果として筆者作のプログラムが混ざるわけですが、そのOSの著作権表示に筆者を含める必要はありません。あなただけが作ったことにしていいです。そのOSを売っていただいて、大金持ちになってもらってもいいです。そうなったときに、分け前をくださいなんて言いませんから、心配しないでお金持ちになってください(*)。

これはこの本を買ってくれた人だけの特権ではなくて、図書館から借りてきたとか、友達に本を貸してもらったとか、なんなら本屋さんで立ち読みした人だって、OKです。そりゃまあ本を買っていただいたほうが、筆者も出版社も助かりますけどね（笑）。

そうやって流用する場合の注意点としては、そのOSの名前は、もう筆者作のOSではないのですから、まぎらわしい名前にはしないでください。それだけがお願いです。どんなに内部のプログラムが似ていても、あなたが、あなたの責任でリリースした、とにかく別のOSなんです。しっかりと立派な名前をつけてあげてください。

以上の扱いは、この本の中で出てくるプログラムとCD-ROMに収められた教材OSプログラムだけです。本書の本文とCD-ROM内のその他のツールを、複製したり改変したりすることは、著作権法によって制限されています。法律の範囲内で取り扱ってください。CD-ROM内のツール類に関するライセンスは、最後の章に書きたいと思います。

8 今後の話の流れ

目次を見るとだいたい分かると思いますが、目次も項目が多いので、ここでおおざっぱな流れをまとめておきます。本文を読みながら、次はどうなるのかなあ、先が楽しみだなあ、という「ワクワク・ドキドキ」したい派は、この部分は読み飛ばしましょう（笑）。この部分は、読み進めているうちに、こんなので本当に大丈夫なんだろうかと思ったら、真っ先に読み返すところだと思ってください。

最初の1週間（1〜7日目）

最初はとにかく、「電源を入れたら自動で動くプログラム」をなんとかして作ります。この部分はC言語で書くのが難しいので、主にアセンブラで作ることになります。

著作権とお金：いやどうしても著作権表示に筆者を加えたいのだ、ということであれば、加えていただいてもいいですよ、もちろん。また途方もなく儲かってしまい、どうしてもお金をあげたいのだ、ぜひもらってくれー、ということであれば、ありがたくいただきますよ（笑）。

それが終わると、今度はディスクからOSを読み出すプログラムを作ります。PCの電源を入れたときには、OSの全体を自動では読み込んでくれないのです。読まれるのは、ディスクの最初の512バイトの部分だけなので、残りを読み込むプログラムを作ります。これもアセンブラで作ります。

そこまでできてしまえば、ここからは先はC言語が使えるので、さっそくC言語を使いながら、画面表示について学びます。同時にC言語の文法にも慣れていきます。このへんは、やりたいことをしているように見えて、実はC言語に振り回されている時期です。

次はマウスカーソルを動かしたいという野望のために、CPUの細かい設定をやり、割り込みルーチンの書き方を覚えます。ここは、本書全体で見てかなりレベルが高いところで、筆者としても少々申し訳ないのですが、しかし話の流れからして、やはりここでやるべきだと思うので、辛抱してください。ここでは、CPUの仕様に振り回され、さらにPCの複雑な仕様にもいじめられます。言語もCとアセンブラが混ざってくるので、これまた混乱が増します。やりたいことをしているなんて、ちっとも思えません。どう見ても「作らされている」期間です。

この苦しい期間を抜けた瞬間が、最初の1週の終わりに当たります。

第2週（8〜14日目）

1週で苦労したかいがあって、気がつけば、たくさんの財産を手に入れています。C言語の基本文法もほとんどマスターし終わって、アセンブラもこの本の要求レベルに到達してしまっています。

そんなわけで、OSっぽい開発がはじまります。しかし今度はアルゴリズムで泣かされます。プログラミング言語の文法が一通り分かっていても、よいアルゴリズムが分からないと、思い通りのOSは作れません。ということで、それを学びながら、OSをゆっくり作っていくことになります。でもこのへんは、やらされている感じはほとんどなくなっています。

第3週（15〜21日目）

さあこれでだいぶ技能が発達しました。もうやりたい放題にOSを作ります。まずはマルチタスクですね。そしてコンソールを作り、ついにアプリケーションにも手を出し始めます。この週の終わりには、完全ではないにしても、もうOSと呼べそうなものができているはずです。

第4週（22〜28日目）

ひたすらOSを多機能にして、それに併せてサンプルアプリケーションをたくさん作っていく段階です。急速に完成度が上がっていきます。一番楽しい時期でもあるでしょう。このへんは説明することも少な

くなって、筆者は本文の日本語で苦労することなく、プログラミングに集中できます（笑）。ああ、日本語といえば、OSが日本語表示に対応するのもこの頃ですね。

おまけの2日間（29〜30日目）

残された2日間で、いろいろ仕上げをします。まだやってなくて、でも簡単にできて面白そうなことは、ここで全部やってしまいます。

■■■■■

以上が1日目から30日目までの内容です。紹介がだんだん短くなっていますが、つまりそれだけ最初の内容が複雑なんです。……それでは、気を引き締めて、1日目へ。あー、緊張しないでいいんです。はい、リラックスぅ。

Chapter 1

一日目

PCの仕組みからアセンブラ入門まで

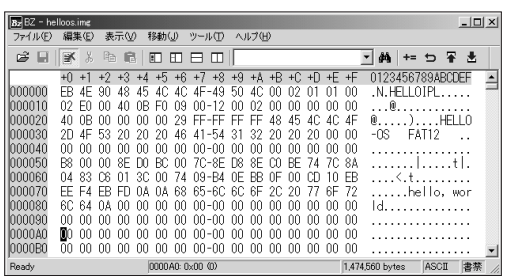
- とにかくやるのだぁ
- 結局何をやったのだろうか？
- アセンブラ初体験
- もうちょっと書き直してみる

1 とにかくやるのだぁ

さてうだうだと話を書くよりも、実際に作るほうがラクなので、さっそく開発です。しかもいきなり今までの説明を無視するかのように、C言語でもアセンブラでもない、全然違ったツールで開発を始めます（笑）。

■■■■■

世の中には「バイナリエディタ」というものがありまして、直訳すると二進数編集機なのですが、これを使って右のような内容のファイルを作りたいのです。



helloos.imgをBZで開いた写真

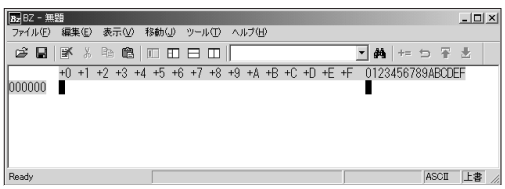
こんなツールは見たことないよ〜、という人もいるかもしれないので、もうちょっと詳しく説明します。

まず、

http://www.zob.ne.jp/~c.mos/soft/bz.html

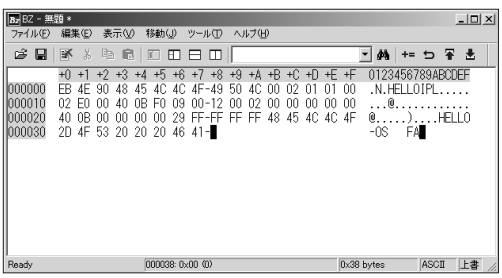
へ行って、「ダウンロードはこちら(Download)」と書いてあるところをクリックして"Bz162.lzh"というファイルをもらってきます（こんなによくできたソフトを無償で公開してくれているc.mosさんに感謝！）。この本が出る頃にはもっとバージョンが新しくなっているかもしれないので、ファイル名は少し違うかもしれません。これを展開してインストールします。そしてさっそくBz.exeをダブルクリックして起動してください。もし起動がうまくいかないようでしたら、上記ページの「★注意★」のところを読んで指示に従ってください。

無事に起動するとこんな画面になります。



BZ起動写真

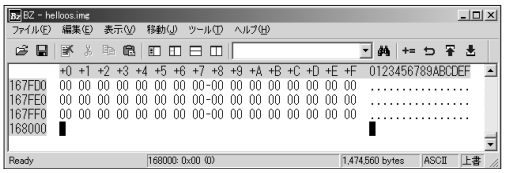
さて、さっそく入力していきます。キーボードから、EB4E904845...と入力していけばいいのです。簡単ですね。間のスペースは自動で入りますので、入力しなくていいです。また右の.N.HELLOIPL.....とかについても、入力しなくていいです。自動で埋まっていきます。もしかしたらバージョンやモードによっては、この右側の表示が写真とは違うかもしれません。その表示はおまけみたいなものなので、とりあえず違っても気にしなくていいです。



000037くらいまで入力したところの写真

000090以降はずっと00です。なんと168000まで入力します。「0」を押していれば勝手にどんどん入っていきますが、結構時間がかかります。もしおうちにネコさんがいましたら、このボタンを押しておいてねって頼んでみてください（猫の手も借りたい）。セロテープとかで押しっぱなしにするとい

う方法もあります。

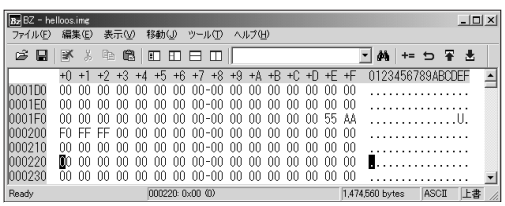


168000周辺の写真

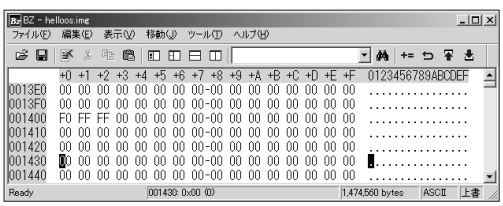
一度に入力できなくて、入力が終わる前にいったん保存しようと思うこともあるかもしれません。そのときは、「ファイル(F)」→「名前を付けて保存(A)」で保存ダイアログが出ますので、適当に保存してください。名前は「helloos.img」がおすすめです。保存したところから再開するときは、まずBz.exeを起動して、「ファイル(F)」→「開く(O)」でファイルを選んで再開すればいいのですが、このとき続きを入力しようとしても無視されます。おおこれはなんということだ！　一気に入力しなければいけなかったのか！　といういうわけではなくて、「編集(E)」→「リードオンリ(R)」をクリックすれば入力できるようになりますので、続きを入れてください。

えーと、うちのネコは気まぐれだし、セロテープもいやだという場合は、適当にマウスで範囲を選択して、「編集(E)」→「コピー(C)」とかができますので、コピー&ペーストをうまく繰り返してすぐに終わらせることもできます。便利な世の中になりましたねえ（しみじみ）。

あ、そうだ。大事なことを忘れていました。0001F0周辺と001400周辺に、00だけではない部分が少しあります。ここのところはそのようにいじってください。そして入力ミスがないかどうかを、全体に渡ってじっくり確認してください。

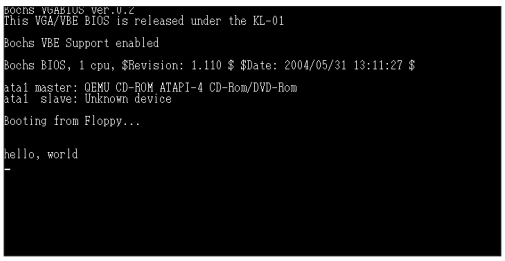


0001F0周辺



001400周辺

それでこれを保存しますと、おお、ディスクイメージ完成です。ファイルのプロパティを見ると、1,474,560バイト（＝1440×1024バイト）と出るはずです。これをフロッピーディスクに書き込んで（後述）して、そのディスクからPCを起動すると、



最初のOS？

というように、「hello, world」と表示されるのです。とりあえずそれだけなんですが、とにかく電源ONと同時に自動で起動して、メッセージを表示するというところまではできたわけです。やったぜー。ちなみに終了方法はありませんので、ディスクを抜いてからPCの電源を切るとか、リセットボタンを押すとかしてください。

■■■■■

さて肝心のディスクに書き込む方法ですが、筆者がそのためのプログラムを用意しておきました。その使い方ですが、それは筆者のツールを一通りインストールしてしまえば説明がラクになるので、まずはインストールの説明からいきます。

付録のCD-ROMを見ると、tolset(*)という名前のフォルダがあると思います。これをコピーして、ハードディスクのどこかへ貼り付けてください。内容は3MB程度ですが、このフォルダの中に自分の作ったものを入れていくことになりますので、あとで大きくなります。だから、100MBくらいの余裕があるところがいいかもしれません。レジストリをいじったりはしませんし、パスの設定とかも不要ですので、インストールはこれだけでおしまいです。あとから丸ごと他のフォルダへ移動させても問題ないです。このツールだけで、OSの開発はもちろんのこと、簡単なWindowsアプリケーションやOSASKアプリケーションなども作れちゃいます。

さてインストールしたtolsetフォルダを開きまして、そこで右クリックして、「新規作成(W)」→「フォルダ(F)」と操作します。「新しいフォルダ」とかいうものができますので、これを「helloos0」という名前に変えて、そのhelloos0フォルダの中に先ほどのhelloos.imgを入れてください。さらに、インストールしたtolsetの中にはz_new_wというフォルダがありまして、その中には!cons_9x.batと!cons_nt.batというファイルもありますので、これらをコピーして、helloos0フォルダの中へ貼り付けておいてください。

tolset：「tool set」の略。道具セット。

さらに、helloos0フォルダの中で右クリックして、「新規作成(W)」→「テキスト ドキュメント」と進んで、ファイル名を「run.bat」にし、「拡張子を変更するとファイルが使えなくなる可能性があります。変更しますか？」に「はい(Y)」と答えて、run.batを作ります。そしてこのrun.batを右クリックして「編集(E)」を選んで

run.bat

```
copy helloos.img ..¥z_tools¥qemu¥fdimage0.bin
..¥z_tools¥make.exe -C ..¥z_tools¥qemu
```

という内容になるように編集してください。

同様に、install.batを作って、

install.bat

```
..¥z_tools¥imgtol.com w a: helloos.img
```

という内容にしておきます。

以上すべてをやり終えたものを、こちらで用意しておきました。お料理番組みたいな感じです。それはCD-ROMのprojectsフォルダの中の、01_dayフォルダの中の、helloos0フォルダです。だからこのhelloos0フォルダをコピーして、tolsetフォルダへ貼り付ければ、いきなり準備完了になります。

■■■■■

さあこのOSもどきを、フロッピーディスクにインストールしてみましょう。100円ショップなどで新品のフロッピーディスクを買ってきて、これをWindowsでフォーマットしておきます（フォーマットは、ドライブにディスクを入れて、マイコンピュータを開いて、「3.5インチFD(A:)」を右クリックして、「フォーマット(A)」を選ぶとできます）。あ、このときにクイックフォーマットは選ばないようにしてください。そしてhelloos0フォルダの!cons_nt.batをダブルクリックすると、コンソールが出てきます（Windows95/98/Meを使っている人は、!cons_9x.batのほうをダブルクリックしてください）。ディスクがドライブに入っているのをよく確認して、そのコンソールに「install」と打ち込むと、インストールが始まります。そのうち終わりますので、終わったらでき上がりです。インストールが終わったら、コンソールはもう閉じていいです。

でき上がったら、そのフロッピーディスクから起動してみてください。きっと、さっきの画面写真のように「hello, world」と表示されます。

注意点を書いておくと、ディスクは新品じゃなくてもいいですが、あまり古いと読み書きがうまくいかないことがあるので、古すぎるものは避けましょう。また、新品でも安物では最初からダメなものがあるので、その場合はあきらめてもう一枚買いましょう。フォーマットやイントールをするとフロッピーディスクは完全に消去されてしまいますから、大事なファイルが入っているディスクで試したりはしないでくださいね。

■■■■■

「説明は分かったけど、わざわざディスクを買ってくるとか、PCを再起動するなんてめんどうだよ、もっと手軽にやる方法はないのか」とか、「そもそもうちのPCはフロッピーディスクドライブがないです」とか、「うちのPCはリセットボタンや電源OFFボタンがないので、こんな変なOSが起動してしまったら、終了方法がないので困りますよー」という人もいるでしょう。そんなこともあるだろうと思って、PCエミュレータを用意しておきました。これを使えば、フロッピーディスクを使ったり、Windowsを終了しなくても、起動したらどうなるかを確認できます。とても便利です。

やり方はとても簡単で、!cons_nt.bat（もしくは!cons_9x.bat）を使って開いたコンソールで、「run」と打ち込んでください。それだけです。QEMUというすばらしいフリーのPCエミュレータがありまして、それが自動で起動します。QEMUは筆者が作ったものではなく、海外の天才的な人たちが作ったものです。感謝です〜。

さて「以上のことをちゃんとやってみたけど、うまくいかないよ〜。うぇーん」という人は、きっとすごく正直な人で、ここまでの説明通り、バイナリエディタで地道にhelloos.imgを作った人だと思います。きっと、どこかで入力ミスをしてしまったのでしょう。どこで間違えたかは分からないのですが、とにかくじっくり000000から000090までと、0001F0あたりを見比べてください。それでもどうしてもダメなときは、めんどくさいのであきらめて、CD-ROMの中に入っている筆者が作ったhelloos.imgを使うことにしてください。

ものぐさな人は、最初からまったく入力しないでhelloos.imgを使ってもいいですが、この体験（苦労して入力して、苦労して間違いを直して、何とか最後に成功すること）は貴重だと筆者は思うので、できれば一度は体験なさることをおすすめします。

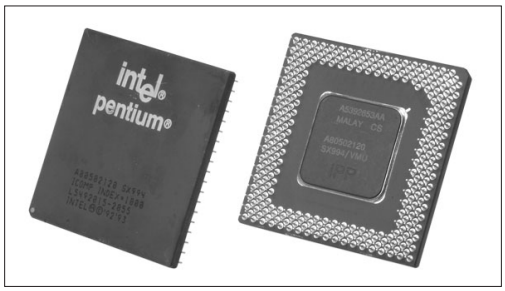
■■■■■

とまあこんなわけで、他のOSを改造することなく、何もないところからOSを作って実行するところまできたわけです（これをOSと認めてくれれば、ですが）。これってすごいことなんですよ。友達に自慢したっていいと思いますよ。開発を始めてからたった数時間で初心者でも一からOSが作れちゃうなんて、この本もなかなかいいじゃないですか（笑）。今回は入力の手間を考えて、メッセージを表示するだけでしたが、もっと長く入力してもよければ、この方法だけでどんなOSでも（まあ1,440KB以内なら）作れます。唯一の問題は、あの「EB 4E 90 48 45 ...」に何の意味があるのか分からないだけです（それが最大の問題でもあるのですが）。で、今日の残りの部分と今後の29日間は、それを解説しているのです。それだけのことなんです。

2 結局何をやったのだろうか？

なんでこんな方法だけでOSができてしまうのか、これを知ることが今後の理解の助けになると思うので、ちょっくら説明させてください。

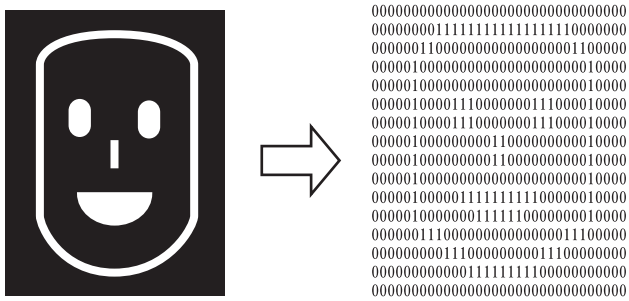
いきなり、PCは結局なんなのかという話から始めるんですが、PCの処理の中心はCPUです。CPUというのは「central processing unit」の略なのですから、これを日本語に直すと「中央処理装置」になって、だから中心なのは当たり前なんです。もし違う装置が中心だとしたら、今度はそれがCPUと呼ばれるだけであって、結局CPUは常に処理の中心なのです。それで、このCPUですが、彼（彼女？）は、電気信号を他の回路とやり取りする以外のことは何もできなくて、しかもその電気信号はONとOFFの2通りしか理解できないという、なかなか無能な人なんです（人じゃないけどね）。



CPU

私たちはPCを使って文章を書いたり、音楽を聴いたり、写真を加工したり、その他さまざまなことをしていますが、いずれの場合もCPUには電気信号が飛び交っているだけで、しかもそれはONかOFFかということだけで、真ん中とか、ちょい上とか、そういう微妙なことはありません。もっと言うと、CPUは文章の内容や音楽のよさや写真の美しさをまったく理解することはなく、この電気信号の組み合わせがきたら、あの電気信号に変換する、などといったことだけをやっております。CPUには計算命令というものがありまして、整数の足し算や引き算はできますし、掛け算や割り算もできます。負の数も使えますし、それだけではなく小数も使えて、10の100乗のような大きな数も扱えて、中学校で教わるような平方根計算や、高校で教わるような対数計算や三角関数すら1命令でこなせるのですが、しかしCPUは数の概念すら理解しておりません。ただこの電気信号に対してこの命令がきたら、その電気信号を出すようにと、回路が組まれているだけです。

話が分かりにくくなってきたので、具体的にいきましょう。たとえばONを1、OFFを0と表すのが一般の人には分かりやすいと思うので、そうしますが、



という32×16＝512個のONとOFFの集まり（＝電気信号の集まり）は、この下手っぴな絵を表しています。一方、

00000000000000000000010010100010

という32個の電気信号の集まりは、1186という整数を表しています（註：1186を2進数で書くと、10010100010になる）。さらに、

01001011010011110100111101000010

という32個の電気信号の集まりは、「BOOK」という単語を表しています（註：これはPC内部で実際に単語を記憶しているときの様子をそのまま書いたもの）。

CPUに見えているものはすべてこのようなものなんです。このようなものしか見えないのです。さらに言うと、

00000100001110000000111000010000

という電気信号を渡されても、これが絵の一部なのか、整数を2進数で表したものなのか、音波の一部なのか、文章の一部なのか、それともゲームのセーブデータの一部なのか、プログラムの一部なのか、そんなことはCPUにはまったく分からないのです。分からないけど、そんなことは気にしないで、ただ黙々と文句もいわずにプログラムにしたがって処理しているだけです。

■■■■■

この話はむしろ逆で、本当はこういう歴史があります。もとをただせば、CPUは電気信号だけを処理する装置として開発されました。そのときはこんなに役立つ機械になるなんて思っていませんでした。しかし、電気信号のON/OFFを数字の0と1に対応づけることで、2進数を電気信号に置き換えられるし、結果の電気信号を2進数に戻すことだってできるということが発見されて、CPUは相変わらず電気信号しか処理できないけれど、2進数計算機としても使えることになりました。

10進数は2進数に直せますし、2進数を10進数に戻すこともできるので、これで普通の計算機も作れるようになりました。また、1つ1つの文字に番号をつけることで（これを文字コードと言う）、文字を数字に対応させられるようになって、これで文字も電気信号に変換できるようになり、CPUに文章処理（たとえばワープロとか検索とか）をさせることもできるようになりました。このあともこの調子で、絵や音楽などを電気信号に対応づける方法が分かって、どんどんCPUの利用範囲が広がっていったのです。しかし相変わらず、CPUは電気信号だけを処理しているのです。

しかもCPUが処理するのはデータだけではないのです。CPUへの命令も電気信号で行います。私たちが作るプログラムは最終的に機械語になりますが、その機械語は電気信号の形でCPUに送られるのです。これは命令番号が連（つら）なっただけのもので、当然のことながら0と1だけのかたまりなのです。

フロッピーディスクは、簡単に言えばこの0と1を磁気のNとSに置き換えた程度のものですし、だからイメージファイルは0と1だけで書けるのです。イメージファイルだけではなく、PCで扱えるあらゆるファイルが0と1だけで書けるのです。むしろ0と1だけで表せないものは、CPUに電気信号の形で渡すことが不可能になるわけで、つまりPCでは処理できないものなのです。

■■■■■

一方、「バイナリエディタ」というのは直訳では「2進数編集機」なわけで、どんな2進数のかたまりだって入力できて、ファイルにできちゃうんです。だから、これは究極の最終兵器で、つまり、バイナリエディタで作れないファイルはないのです（おぉ！）。お店で売っているあのソフトがほしいなあ、でも買えないなあ、なんていうときは、家でひたすらバイナリエディタで入力していけばいいんです。これだけでお店に並んでいたものとまったく違いのないものが自分で作れちゃうんです。500万画素のデジタルカメラが買えなくても気にすることはありません。バイナリエディタがありさえすれば、500万画素のデジタルカメラで撮ったJPEGファイルとまったく同じ画像ファイルが、いくつでも作れちゃうんですから。高価なCコンパイラが買えないよう、と泣くことはないのです。そんなものがなくたって、そのコンパイラが生成したのと同じ結果の実行ファイルを、バイナリエディタだけで作れるのです。なんならそのコンパイラそのものをバイナリエディタだけで作ることもできます。

こんな最強ツールなのですから、OSくらい作るのはわけないのです。そんなわけで、今回あっけなくOSができちゃったのは、当たり前のことなんです。たったこれだけのことを言うために、こんな長い話をする必要があったのかと思うかもしれませんが、CPUの基礎が分かると先々の話の分かりやすさが違いますので、今はガマンしてください。

■■■■■

「おいちょっと待て、バイナリエディタの直訳が2進数編集機だというのはその通りだと思うが、おいらはあんたのhelloos.imgを入力するときに、0と1以外にもいろいろ入力させられたぞ。そもそもの最初からして、Eじゃないかよ。これのどこが2進数なんだよ。これはアルファベットじゃねーか」……おっとこれは失礼しました。おっしゃる通りです。

2進数は、電気信号との相性は抜群なのですが、やたらと桁が多くなるというさみしい欠点がありまして、例としては、1234を2進数で書くと10011010010と、なんと11桁にもなるわけです。10進数ではたったの4桁だったのにねえ。これでは紙がもったいないので、16進数というものをこの業界では使います。16進数では1234は4D2となり、なんと3桁です。

なんでわざわざ16進数などというものを持ち出すのか、10進数のままでいいじゃないかと思うところですが、実は2進数から16進数に直すのが非常に簡単で、

2進数←→16進数の表

0000 - 0	0100 - 4	1000 - 8	1100 - C
0001 - 1	0101 - 5	1001 - 9	1101 - D
0010 - 2	0110 - 6	1010 - A	1110 - E
0011 - 3	0111 - 7	1011 - B	1111 - F

という表を使って、2進数を下の位から4桁ずつ区切って置き換えるだけで、

```
100 1101 0010 → 4D2
```

にできちゃうのです。逆に、4D2からもとの100110100010にするのも簡単で、この逆をやればいいわけです。10進法ではこんなに簡単には2進法に変換できないのです。これと同じ理由で、3桁ずつ区切る8進法もたまに使われています。

そんなわけで、みなさんがEBと入力したときには、11101011という入力がなされたのです。だからまあ16進数エディタではあるのですが、これをバイナリエディタと呼んでいる習慣をおおめに見てください。

■■■■■

バイナリエディタをべたほめの筆者でありますが、何のことはない、これは結局、「紙と鉛筆さえあれば、どんなすごい小説だって書けちゃうのだ」というのと大差ないのであります。紙や鉛筆は、しょせん紙や鉛筆であって、すごい小説を書きやすいというわけではありません。だからみんな、プログラムを作るときにテキストエディタとコンパイラを使うのであって、バイナリエディタを使わないわけです。写真ファイルを作るのもデジカメでやって、バイナリエディタは使わないわけです。そんなわけで、私たちも、バイナリエディタに頼った開発はこのくらいでやめて、プログラミング言語へ移行していくことにします。でもとにかく、この経験を踏まえて、いざというときには、バイナリエディタというものは非常に役に立つものだ、ということを知っておいてください。これからもたまに使いますよ。

3 アセンブラ初体験

さて、ではさっそく、先ほどとまったく同じhelloos.imgを生成するための、アセンブラのソースプログラムを書きましょう。今回使用するアセンブラは、「nask（なすく）」という筆者作のアセンブラです。これは「NASM（なすむ）」という、フリーソフトでは結構有名なアセンブラの文法の多くをまねて、でもNASMよりも自動最適化能力を高めた、そういうアセンブラです。

超長いソースプログラム

```
    DB  0xeb, 0x4e, 0x90, 0x48, 0x45, 0x4c, 0x4c, 0x4f
    DB  0x49, 0x50, 0x4c, 0x00, 0x02, 0x01, 0x01, 0x00
    DB  0x02, 0xe0, 0x00, 0x40, 0x0b, 0xf0, 0x09, 0x00
    DB  0x12, 0x00, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00
    DB  0x40, 0x0b, 0x00, 0x00, 0x00, 0x00, 0x29, 0xff
(ページがもったいないので途中の18万4314行を省略)
    DB  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
```

という超長いソースプログラムをコピー＆ペーストを駆使して作って、これを「helloos.nas」というファイル名でhelloos0の中に保存します。よく見ると分かりますが、これはさっきバイナリエディタで入

力したものがそのまま並んでいるだけです。

それで、!cons_nt.batか!cons_9x.bat（どちらになるかは先ほどと同じように、使っているWindowsのバージョンによりけりです。以後、毎回この説明を書くのはめんどうなので、これを!consと省略して書きます）で開いたコンソールで、

プロンプト(*)>..¥z_tools¥nask.exe helloos.nas helloos.img

と入力すれば、helloos.imgができます（「プロンプト>」の部分は入力しなくていいです）。

やった、初アセンブラ体験だ！ ……が、いくらなんでもこれはひどいと思うんです。18万行を超えるソースプログラムを作るのはとても大変ですし、ハードディスクがもったいないです。こんなことをするくらいなら、バイナリエディタで作るほうが、「0x」とか「,」とかを入力しなくていい分だけ、ラクなのです。

■■■■■

ということで、DB命令だけではなくRESB命令も使うことにして、helloos.nasをぐっと短くしました。短くなったけど、もちろん内容は変わっていません。さあご覧ください。

まともな長さのソースプログラム

```
    DB  0xeb, 0x4e, 0x90, 0x48, 0x45, 0x4c, 0x4c, 0x4f
    DB  0x49, 0x50, 0x4c, 0x00, 0x02, 0x01, 0x01, 0x00
    DB  0x02, 0xe0, 0x00, 0x40, 0x0b, 0xf0, 0x09, 0x00
    DB  0x12, 0x00, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00
    DB  0x40, 0x0b, 0x00, 0x00, 0x00, 0x00, 0x29, 0xff
    DB  0xff, 0xff, 0xff, 0x48, 0x45, 0x4c, 0x4c, 0x4f
    DB  0x2d, 0x4f, 0x53, 0x20, 0x20, 0x20, 0x46, 0x41
    DB  0x54, 0x31, 0x32, 0x20, 0x20, 0x20, 0x00, 0x00
    RESB    16
    DB  0xb8, 0x00, 0x00, 0x8e, 0xd0, 0xbc, 0x00, 0x7c
    DB  0x8e, 0xd8, 0x8e, 0xc0, 0xbe, 0x74, 0x7c, 0x8a
    DB  0x04, 0x83, 0xc6, 0x01, 0x3c, 0x00, 0x74, 0x09
    DB  0xb4, 0x0e, 0xbb, 0x0f, 0x00, 0xcd, 0x10, 0xeb
    DB  0xee, 0xf4, 0xeb, 0xfd, 0x0a, 0x0a, 0x68, 0x65
    DB  0x6c, 0x6c, 0x6f, 0x2c, 0x20, 0x77, 0x6f, 0x72
    DB  0x6c, 0x64, 0x0a, 0x00, 0x00, 0x00, 0x00, 0x00
    RESB    368
    DB  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x55, 0xaa
    DB  0xf0, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00
    RESB    4600
    DB  0xf0, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00
    RESB    1469432
```

これを入力するのはきっとめんどうでしょうから、付録CD-ROMのprojectsフォルダの中の01_dayフォルダの中の、helloos1フォルダに入れておきました。だからこのhelloos1フォルダをコピーして、

プロンプト：コンソールで出てくる、入力を促すためのメッセージ。prompt。

tolsetフォルダへ貼り付けてください。さっきまでのhelloos0フォルダはもういらないので、消しちゃってもいいですが、記念にとっておいてもいいです。ちなみに筆者は、helloos0フォルダをhelloos1に名前変更して、いらないファイルを消して、必要なファイルを新規作成して編集して、helloos1フォルダを作りました。そうやって少しずつ成長させていっているわけです。

毎回アセンブルするたびに、さっきの命令を入力するのはめんどうなので、バッチファイル(*)「asm.bat」も作っておきました。だから!consで開いたプロンプトで「asm」と入れるだけで、helloos.imgが生成されます。「asm」でimgファイルを作って、そのあとに「run」とすれば、先ほどと同じ結果になります。

■■■■■

DB命令というのは「data byte」の略で、つまりファイルの内容を1バイトだけ直接書く命令です。筆者はアセンブラの命令を大文字で書くのが好きなので大文字にしていますが、「db」と小文字にしても問題なくアセンブルされます。

この命令はアセンブラの世界における最終兵器でして、つまりDB命令さえあればどんなデータも（ついでに言うならどんなプログラムも）記述できるわけです。そんなわけで、アセンブラに作れないファイルはありません。テキストファイルだって、画像ファイルだって、やれと言われれば何でも作れるのです。他の言語（たとえばC言語）ではこのような万能性（？）はめったにありません。

RESB命令は「reserve byte」の略で、とりあえずここに10バイトほどあけておいてくれ、みたいな意味で RESB 10 というふうに使います。この10バイトは予約ずみ、というわけです（指定席みたいなものを想像してください）。しかしnaskでは単にあけておくだけではなくて、あけた部分を0x00で埋めるということになっていますので、今回は0x00をたくさん書き込む代わりに使っています。この命令のおかげで18万行も書かなくてすんでいるのですから、大助かりです。

数値の前に「0x」をつけると16進数で、つけなければ10進数、というのはC言語と同じです。

4 もうちょっと書き直してみる

わずか22行になったのはまことに嬉しいわけですが、やっぱり意味不明な雰囲気に変わりはないわけで、それをもうちょっと意味ある状態に書き直してみます。ソースファイルは48行に増えてしまいました。できたファイルはCD-ROMの中の projects/01_day/helloos2 にありますので、helloos2フォルダをtolsetの中へコピーして準備完了にしてください。もうhelloos1フォルダも消しちゃっていいです（今後、いちいち書きませんが、それぞれのフォルダは独立していますので、使い終わったものは削除してしまってかまいません。もちろん記念にコレクションしておいてもかまいません）。

バッチファイル：基本的には、コンソールに入力するものをテキストファイルに書き並べただけのもの。もっと高度なこともできるのだが、ここではそこまでやってない。バッチというのはバッチ処理のことで、これは一連の作業を一気にやることを意味する。batch file。

50行もあるのですが、これはページをさいても損はないと思うので載せておきます。

かなりそれっぽいソースプログラム

```
; hello-os
; TAB=4

; 以下は標準的なFAT12フォーマットフロッピーディスクのための記述

        DB      0xeb, 0x4e, 0x90
        DB      "HELLOIPL"      ; ブートセクタの名前を自由に書いてよい（8バイト）
        DW      512             ; 1セクタの大きさ（512にしなければいけない）
        DB      1               ; クラスタの大きさ（1セクタにしなければいけない）
        DW      1               ; FATがどこから始まるか（普通は1セクタ目からにする）
        DB      2               ; FATの個数（2にしなければいけない）
        DW      224             ; ルートディレクトリ領域の大きさ（普通は224エントリにする）
        DW      2880            ; このドライブの大きさ（2880セクタにしなければいけない）
        DB      0xf0            ; メディアのタイプ（0xf0にしなければいけない）
        DW      9               ; FAT領域の長さ（9セクタにしなければいけない）
        DW      18              ; 1トラックにいくつのセクタがあるか（18にしなければいけない）
        DW      2               ; ヘッドの数（2にしなければいけない）
        DD      0               ; パーティションを使ってないのでここは必ず0
        DD      2880            ; このドライブの大きさをもう一度書く
        DB      0,0,0x29        ; よく分からないけどこの値にしておくといいらしい
        DD      0xffffffff      ; たぶんボリュームシリアル番号
        DB      "HELLO-OS   "   ; ディスクの名前（11バイト）
        DB      "FAT12   "      ; フォーマットの名前（8バイト）
        RESB    18              ; とりあえず18バイトあけておく

; プログラム本体

        DB      0xb8, 0x00, 0x00, 0x8e, 0xd0, 0xbc, 0x00, 0x7c
        DB      0x8e, 0xd8, 0x8e, 0xc0, 0xbe, 0x74, 0x7c, 0x8a
        DB      0x04, 0x83, 0xc6, 0x01, 0x3c, 0x00, 0x74, 0x09
        DB      0xb4, 0x0e, 0xbb, 0x0f, 0x00, 0xcd, 0x10, 0xeb
        DB      0xee, 0xf4, 0xeb, 0xfd

; メッセージ部分

        DB      0x0a, 0x0a      ; 改行を2つ
        DB      "hello, world"
        DB      0x0a            ; 改行
        DB      0

        RESB    0x1fe-$         ; 0x001feまでを0x00で埋める命令

        DB      0x55, 0xaa

; 以下はブートセクタ以外の部分の記述

        DB      0xf0, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00
        RESB    4600
        DB      0xf0, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00
        RESB    1469432
```

■■■■■

新しい見どころとしては、まず「;」命令ですね。これはコメント命令で、C言語やC++で言うところの「//」です。この命令のおかげで、ソースプログラムにいろいろとコメントをつけられます。

次はDB命令の新しい使い方です。なんと文字列を書くことができます。文字列を書くと、文字列を構成するそれぞれの文字の文字コードを調べて、それを1バイトずつ並べてくれます。これは便利です。というのは、たとえば出力するメッセージを変更したいときに、文字コード表とにらめっこしなくていいからです。

あとはDW命令とDD命令ですね。これらはそれぞれ「data word」「data double-word」の略で、DB命令の仲間です。ワードというのは、直訳すると「単語」という意味ですが、PCのアセンブラの世界でワードというのは、16ビットのことなのです。2バイトでもあります。ダブルワードというのは32ビットのことで、バイトで言えば4バイトです。

おっと「RESB 0x1fe−$」の説明を忘れていました。このドルマークが非常に意味不明だと思いますが、これはこの行が先頭から何バイト目かを教えてくれる変数です（厳密に言うと違う意味になるときもあるのですが、その話は明日）。このプログラムでは、これに先立って132バイト分を記述する命令があるので、ここでの$は132です。そんなわけで、naskは0x1feから132を引いて378という答えを得て、だから378バイトの00を並べてくれるわけです。

それなら$なんて使わないで直接378と書けばよさそうなものですが、メッセージを "hello, world" から "This is a pen." に変更した場合、ここにいくつの00を入れるかが変わってきます。というのは、フロッピーディスクの510バイト目に（つまり0x1feバイト目に）、55 AAがきてくれないと困るからです。ドルマークを使ってプログラムを書いておけば、いくつの00を入れなきゃいけないのかという計算は全部アセンブラがやってくれますので、メッセージを気軽に改造できるというわけです。

■■■■■

これでメッセージを改造しやすくなったので、ぜひお気に入りのメッセージを表示するOSに改造してください。そうすることで、世界に1つだけの、あなただけのOSになります。しかし残念なことに、日本語などの全角文字は出力できません。やってみても害はないですが、文字化けしてしまうのです。これはこのプログラムが全角文字を表示できないせいですので、今のところは英語かローマ字でガマンしてください。

■■■■■

最後にソースプログラム中に出てきた専門用語などを解説しておきます。さあて、夜もふけてきたので、今日はここまで。その他の説明とかはまたあしたー。

TAB=4 ……タブサイズが設定できるテキストエディタを使っている人は、タブサイズを4に設定してください。ソースが読みやすくなります。メモ帳はタブサイズが8に固定なんですよねー。よし、明日はおすすめのテキストエディタの紹介からやりましょう。

FAT12フォーマット ……WindowsやMS-DOSでフロッピーディスクをフォーマットすると、この形式でフォーマットされます。hello-osでもこの形式を採用して、その中にOSを押し込むことにしました。そうすることで、このディスクを入れてもWindowsに怒られないですみますし、あいたディスク領域に好きなファイルをしまっておくこともできます。FAT12 format。

ブートセクタ ……フロッピーディスクの最初のセクタのことをこう呼びます。そもそも「セクタって何よ？」なんですが、フロッピーディスクは1バイトずつの読み書きができない仕組みになっていまして、512バイトずつまとめて読み書きしています。そんなわけで、フロッピーディスクの世界では、512バイトを1セクタと呼ぶのです。フロッピーディスクは1,440KBで、つまりは1,474,560バイトなのですが、これを512で割ると2,880になるので、フロッピーディスクは合計2,880セクタで構成されているわけです。で、どうして最初のセクタがブー

トセクタと呼ばれるのかと言いますと、PCはまずディスクの最初のセクタを読むんです。それで最後の2バイトを見ます。これが 55 AA でなければ、このディスクは起動に必要なプログラムは書かれていないようだ、と判断し、起動できないよエラーがでます（なんで 55 AA なんだよー、とか思うかもしれませんが、それはPCを設計した人たちが勝手に決めたことなので、筆者に聞かれても困ります）。もしめでたく 55 AA が確認できた場合は、このセクタの先頭からプログラムが書いてあるだろうということで、そこから実行し始めてくれます。boot sector。

IPL ……initial program loaderの略。初期プログラム読み込み機。ブートセクタはたったの512バイトしかないわけで、hello-os以外の普通のOSは到底このサイズには収まりません。そんなわけでほとんどすべてのOSは、ブートセクタにOS本体を読み込むプログラムを組み込んでいます。そういう事情があるので、ブートセクタのことをIPLと呼ぶことがあるのです。でもhello-osではプログラムをロードする機能はないわけで、だから「HELLOIPL」っていう名前はちょっと変ですね。こんな嘘つきは許さないぞ！　という正義感あふれる方がおられましたら、なんか違う名前に直してください。必ず8バイトにしないといけないので、もし7バイト以下になったらスペースをうしろにつけておいてください。

ブート ……起動のこと。そもそもブートというのは、ブーツ（長ぐつ）の単数形なのですが、それとPCの起動とどういう関係があるのでしょうか。本来なら普通にstartと言うべきところなのです。実はこのブートはブートストラップ（bootstrap）の略でして、これは本来はブーツについている「つまみ皮」のことなんです。しかし「ほらふき男爵の冒険」という物語以降、この物語にちなんでbootstrapという単語には「自力で成し遂げる」という意味を持つようになりました（このへんの詳しいいきさつは、Googleで調べるか、もしくはサポートページ（http://hrb.osask.jp/）で質問してください）。それで、ディスクにOSが入っているのに、そのOSを読み込むプログラム（つまりIPL）もディスクに入っているんですよーという、まるで「宝箱を開けるための鍵は宝箱の中に入っています」みたいな、なんか一種矛盾しているかのような、このOSの自力起動の仕組みを、bootstrap方式と呼ぶようになったのです。bootという言い方はそれにちなんでいるわけです。まあ筆者だったらbootstrap方式なんていう怪しい名前にはしないで、「多段ロケット方式」と命名したでしょうね。

Chapter2

二日目

アセンブラ学習とMakefile入門

- まずはテキストエディタの紹介
- さて開発再開
- ブートセクタだけを作るように整理
- 今後のためにMakefile導入

1 まずはテキストエディタの紹介

筆者がおすすめのテキストエディタは、TeraPadというもので、

http://www5f.biglobe.ne.jp/~t-susumu/library/tpad.html

に行くとダウンロードできます。フリーソフトです（寺尾進さんに感謝～！）。

このページの「tpad089a.exe」というファイルをダウンロードして実行すると、インストールできます。例によってこの本が出る頃にはバージョンアップされているかもしれません。その場合はファイル名が少し違うかもしれないので注意してください。インストール後にTeraPad.exeを実行すると、あとは基本的にメモ帳と同じように使えます。いろいろ設定があるので好みに合わせてほしいのですが、とりあえず筆者のおすすめ設定を紹介しておきますね。

「表示(V)」→「タブの文字数(A)」という項目を選ぶと設定項目が出てきますので、これを4にします。さらに、「表示(V)」→「オプション(O)」を選ぶとダイアログが出てくるので、「ルーラ/行番号」という見出しを選んで、「標準モード時は非表示(V)」のチェックを外します。また「論理行で行番号を表示する(I)」

にはチェックをつけます。今度は「表示」という見出しを選んで、マーク項目のところを見て、改行や[EOF]が表示されているほうがいいかどうかを選ぶことができます。筆者はどちらも表示されないほうがすっきりしていて好きなので、どちらのチェックも外していますが、ここは好みが分かれるところなので、いろいろ試してみてください。以上の設定が終わったら、「OK」ボタンを押してダイアログを閉じます。これでおすすめの設定は完了です。

あー、なんか昨日からいろいろフリーソフトを紹介していますが、もしすでにお気に入りのバイナリエディタやテキストエディタを持っているのでしたら、それを使ってもらってかまいません。違うソフトを使っても、プログラムの内容には影響しませんので、CD-ROMにも入れていません。筆者のおすすめなんて気にしないで、好きなのを好きなように使ってください。

2 さて開発再開

昨日はhelloos.nasのコメント部分について十分に説明し終わっていなかったのですが、「プログラム本体」より前の部分と「ブートセクタ以外の部分」については、フロッピーディスクに関するある程度詳しい知識が必要で、それはあとになったら出てくるので、それまでは保留ということにします。

そうなると意味不明なのはプログラム本体だけなのです。ということで、どんどん分かりやすい形に書き直しちゃいましょう。projects/02_day のhelloos3をtolsetの中にコピーして、その中のhelloos.nasを開いてみてください。以下に途中までを書きますね。

helloos.nasの途中まで

```
; hello-os
; TAB=4

        ORG     0x7c00          ; このプログラムがどこに読み込まれるのか

; 以下は標準的なFAT12フォーマットフロッピーディスクのための記述

        JMP     entry
        DB      0x90

--- (中略) ---

; プログラム本体

entry:
        MOV     AX,0            ; レジスタ初期化
        MOV     SS,AX
        MOV     SP,0x7c00
        MOV     DS,AX
        MOV     ES,AX

        MOV     SI,msg
putloop:
        MOV     AL,[SI]
        ADD     SI,1            ; SIに1を足す
        CMP     AL,0
```

```
        JE      fin
        MOV     AH,0x0e         ; 一文字表示ファンクション
        MOV     BX,15           ; カラーコード
        INT     0x10            ; ビデオBIOS呼び出し
        JMP     putloop
fin:
        HLT                     ; 何かあるまでCPUを停止させる
        JMP     fin             ; 無限ループ

msg:
        DB      0x0a, 0x0a      ; 改行を2つ
        DB      "hello, world"
        DB      0x0a            ; 改行
        DB      0
```

今回は新出命令がたくさんです。上から順番に見ていきましょう。

■■■■■

最初はORG命令です。これは、この機械語が実行時にPCのメモリのどこに読み込まれることになるのかをnaskに教えてあげるための命令です。これがないと、いくつかの命令は正しく翻訳できないのです。またこの命令を書くと、ドルマーク（$）の意味も変わりまして、出力ファイルの何バイト目であるかではなく、読み込まれる予定のメモリの番地になります。

ORG命令の元になった英語は「origin」で、意味は開始点です。ここからプログラムが始まるよー、ちなみにここはメモリの○×番地に読み込まれる予定なんだよー、そういうことだから、naskくん、あとはよろしくねー、みたいな感じです。ここでは0x7c00という番地を指定していますが、その理由はあとで（この節の終わりのほう）。

その次に出てくるのはJMP命令です。これはC言語で言うところのgoto文です。もとになった英語は「jump」です。ジャンプ〜。簡単ですね。

その次が entry: で、これはラベルの宣言です。JMP命令の飛び先の指定などに使います。C言語とよく似ています。ちなみにentryは「入り口」という意味です。

■■■■■

次は……MOV命令のようです。MOV命令は多分一番よく使う命令で、ここでもDB命令の次に多い命令になっています。とても単純な命令で、ようするにただの代入です。単純な命令ですが、このMOVを完全にマスターすれば、アセンブラの半分以上は理解できたことになると、筆者は思っています。だからていねいに説明しますよ。

MOV AX,0 というのは、AX = 0; という代入文です。同様に、MOV SS,AX というのは、SS = AX; という代入文です。「ところで、このAXやSSというのは何ですか？」という声が聞こえてきますが、それはちょっと待ってくださいね。

MOV命令の元になった英語は「move」で、移動という意味なのですが、代入って移動とは似ているけどちょっと違いますよね。移動したら移動元は空っぽになるわけですよ、普通。でも、MOV SS,AXを実行してもAXが空っぽになったりはしません。そのままの値です。だからやっぱり代入なんです。すなおにコピー命令とかだったら説明が簡単だったのになあ。どうしてMOV命令ということになったのかは、筆者にも分かりません。

■■■■■

さて、AXやSSの説明です。CPUにはレジスタという記憶回路がありまして、これはまさに機械語における変数です。代表的なレジスタは次の8つです。それぞれのレジスタの名前には本名がありまして、それを知る機会が今ではあまり多くないようなので、ついでに紹介しておきましょう。

AX ……アキュムレータ（accumulator：累積演算機という意味）
CX ……カウンタ（counter：数を数える機械という意味）
DX ……データ（data：データという意味）
BX ……ベース（base：土台とか基点という意味）
SP ……スタックポインタ（stack pointer：スタック用ポインタ）
BP ……ベースポインタ（base pointer：ベース用ポインタ）
SI ……ソースインデックス（source index：読み込みインデックス）
DI ……デスティネーションインデックス（destination index：書き込みインデックス）

これらはすべて16ビットレジスタで、だから16桁の2進数を記憶できるわけです。それぞれには上に示した通りたいそうな本名がありますが、「エイエックス・レジスタ」とか「エスアイ・レジスタ」など、アルファベットをそのまま読むのが普通のようです。

それでも、この本名はちゃんと意味があります。たとえば、この8つのどのレジスタを使ってもたいていは同じ計算ができるのですが、各種の演算にはAXを使うようにすると、プログラムが簡潔（かんけつ）に書けるのです。一例を挙げれば、

ADD CX,0x1234　は　81 C1 34 12　という4バイトの命令ですが、
ADD AX,0x1234　は　05 34 12　　という3バイトの命令なのです。

この例で分かると思いますが、簡潔に書けるというのは、あくまでも「機械語で書いたら」という意味であって、アセンブラのソースプログラムの段階では見た目の差はありません。機械語が分からなければ、理解しにくいことって結構あるのです。

他のレジスタについて言えば、CXは回数を数えるのに使うと便利になるようにできていますし、BXはメモリの番地計算の基点に使うと便利になるようになっています。他のものについても同様です。

AX，CX，DX，BXの名前の由来には、Xに当たる略語が見当たらないのですが、このXは拡張

(extend) という意味です。一体何が拡張なのかというと、昔はCPUでレジスタといえば8ビットが主流だった時代がありまして、それと比べると16ビットのレジスタというのはすごいことだったのです。それで「拡張だぜすごいだろう」ってことで、Xがついているのです。レジスタの並びがアルファベット順になっていませんが、これは機械語におけるレジスタ番号の順に並べたためで、筆者の気まぐれというわけではありません。

なお、8個のレジスタを全部合わせても16バイトにしかなりません。つまりこのレジスタを全部使っても、CPUはたったの16バイトしか記憶できないわけです。

一方、CPUには8ビットのレジスタも8個あります。

AL ……アキュムレータロウ (low：低い)
CL ……カウンタロウ
DL ……データロウ
BL ……ベースロウ
AH ……アキュムレータハイ (high：高い)
CH ……カウンタハイ
DH ……データハイ
BH ……ベースハイ

なんか名前がちょっと似ていますが、これは理由がありまして、AXレジスタの16ビットのうち、下の位 (くらい) のビット0からビット7の8ビットの部分をALと言うのです。また、上の位のビット8からビット15の、つまり残りの8ビットの部分をAHと言うのです。そんなわけで、レジスタが増えた、これでさらにCPUだけで8バイトも余計に記憶できるぞ、ばんざーい、と思ったらそうではなくて、あいかわらず合計16バイトしか記憶できないわけです。CPUはかなり記憶能力がないようです。

ちなみにBP, SP, SI, DIはLとHに分けられないのかな？　と思ったあなたは実にいいセンスですが、分けられないのです。どうしても分けた結果が必要なときは、MOV AX,SI とかやって、その上でALとAHを使ってくれ、というのがインテルの設計者のお考えのようです。

「おい、うちのPCは16ビットじゃないぞ、32ビットなんだぞ、32ビットっていうのは、データの処理が32ビットずつできることなんだ

ろ？　32ビットのレジスタはどこにあるんだ？」という疑問が、そろそろ出てくると思うので、お答えします。

EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI

これが32ビットレジスタです。今回のプログラムでは一度も出てきていませんが、しかし使いたければ問題なく使えます。レジスタ名は、16ビットレジスタ名の頭にEをつけただけでして、このEの由来は何かというと、やっぱりextendなわけです。16ビット時代から見れば、32ビットは大拡張だったわけです。EAXは32ビットレジスタなのですがこれはやっぱりというか、毎度のことというか、AXと一部共通になっていまして、32ビットのうちの下の16ビットがまさにAXそのものです。上の16ビットは何と呼ぶのかと言いますと、なんと名前がありません。レジスタ番号もありません。つまり、EAXを16ビット2つに分けて使うことはできるのですが、簡単に使えるのは下のほうだけで、どうしても上のほうを使いたければ、たとえば16ビットずらす命令を使って、上の16ビットを下へおろしてこないといけないのです。

とまあそんなわけで、32ビットになってもCPUは32バイトしか記憶できないわけです。かなり記憶能力ないです。

読者の中には64ビットのPCを使っている人もいるのではないかと思いますが、今回は64ビットレジスタが使えるモードを利用しないので、説明はここまでにいたします。

これでレジスタの紹介は終わり、としたいところなんですが、ああそうか、まだいたかー。今度はセグメントレジスタというレジスタの紹介です。どれも16ビットのレジスタです。

ES ……エクストラセグメント（extra segment：おまけセグメント）
CS ……コードセグメント（code segment）
SS ……スタックセグメント（stack segment）
DS ……データセグメント（data segment）
FS ……本名なし（おまけセグメントパート2）
GS ……本名なし（おまけセグメントパート3）

これらのセグメントレジスタについては、今は説明しません。ごめんなさい。明日説明しますので、そのときまでおあずけです。とりあえず、どれも0を入れておけば問題ないんです、今のところは。

よし、これでレジスタの説明は十分です。

■■■■■

さてプログラムを読み進めていきます。次に意味が分からないのは、MOV SI,msg ですね。MOVは代入なので、SI = msg; という意味なのですが、msgというのは下に出てくるラベルでして、「ラベル

をレジスタに代入するなんて、一体どういうことだ！」というわけです。

この謎を理解するために、あえてJMP命令に戻ってみましょう。JMP entry という命令がありましたが、実はここを JMP 0x7c50 と書いてもまったく問題ありません。もともとJMP命令というのは、メモリの番地を指定するのが基本形で、だからこれは0x7c50番地のプログラムへ飛べ、という意味なのです。

なぜ JMP entry が JMP 0x7c50 の代わりになっているのかと言えば、それは、entryが0x7c50だからです。アセンブラでは、ラベルはすべてただの数字なのです。どんな数字になるかというと、ORG命令をもとに「この部分は、メモリの○×番地に読み込まれることになるはず」を計算し、その値をラベルの値ということにしてくれるのです。

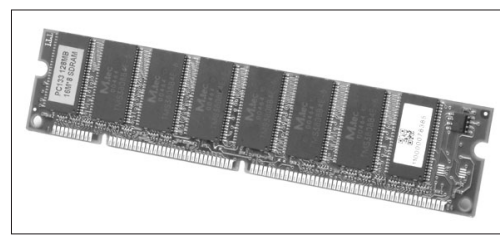
だからもし、このプログラム中で MOV AX,entry と書いたら、AXには0x7c50が代入されることになります。AXに入るのは、このようなただの数字であって、entry以降に書かれたプログラムが記憶されるとか、そんなとんでもないことが起きたりはしません。

では MOV SI,msg はどうなるのかというと、ここではmsgは0x7c74になっているので、SIには0x7c74が代入されることになります。

■■■■■

お次は MOV AL,[SI] です。MOV AX,SI とかだったら説明はいらないのですが、[ ]がついています。アセンブラではこれがつくと意味が完全に変わってしまうので、じっくり説明したいと思います。

この記号は「メモリ」を意味します。メモリというのは、PCを自作したことがあればすぐに分かると思いますが、256MBとか512MBとかの、あの部品のことです。



メモリ

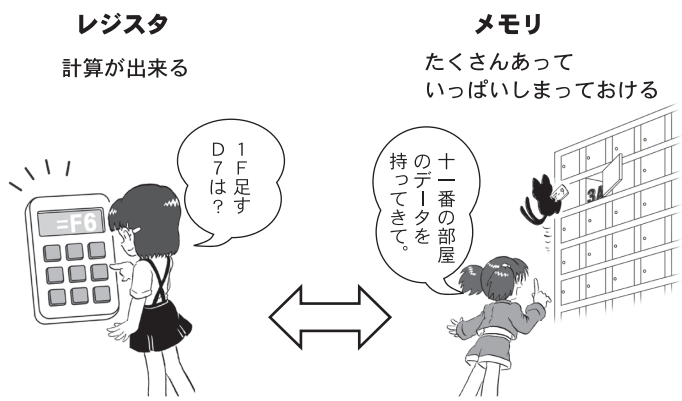
今までほとんどまともな説明もないまま、メモリ、メモリと言ってきましたが、これは一体なんでしょうか。はい、ちょー大規模な記憶素子団地です。いやほんとに団地という表現がぴったりなくらい、びっちりと、そして非常に規則正しく、記憶素子たちは並んでいます。記憶のことを英語ではmemoryと言いまして、だからこの素子もメモリと呼ばれているわけです。ああ、きっと忘れていそうだから言いますが、読むときは「メモリー」ですよ。

レジスタの説明でCPUの恥ずかしい記憶能力が明らかになった今、まともな情報処理をするには、記憶する回路を別に用意してやらないといけないのは自明のことです。32ビットCPUでさえ、普通のレジスタだけでは32バイトしか覚えられないわけです。あのよく分からないセグメントレジスタをフルに使ったとしても、合計たったの44バイトです。これじゃあ、実行に必要なブートセクタの中身すら満足に覚えられません。

さて記憶素子が不可欠であるということは分かってもらえたとして、先に進みますが、メモリはCPUの中ではなく、外にあります。だからCPUからすると外部記憶装置です。これは意外に重要なことでして、つまりCPUは自分の足（端子のこと）の一部を使って、メモリに向かって「おーい、5678番地のデータを僕の足へ送ってくれよー」という電気信号をメモリへ送るわけです（厳密には間にチップセットという制御素子が入りますが）。CPUがメモリのデータを読んだり書いたりするときは、まさにこういうやり取りが行われているわけです。

メモリと交信するのはデータをやり取りするときだけではありません。そもそもプログラム自体もメモリのどこかに必ずあります。プログラムというのは一般的に言って44バイトに収まらないのが普通ですから、レジスタに入るわけがないのです。プログラムは必ずメモリに置く、という規則になっています。CPUが機械語を実行するときには、必ずメモリからプログラムを1命令ずつ読み出して、順番に実行しているわけです。

こんなに大事なメモリくんなのに、CPUからは結構離れた位置にいます。いやもちろん10cmくらいの距離かもしれませんが、これはCPUの半導体の中よりはずっとずっと遠いわけです。ということで、データを教えてくれよーとか、このデータを僕の代わりに覚えておいてくれよーと言ってから、メモリが無事にその要求にこたえるまでに、結構時間がかかるんです（CPUはとても高速に動作していますので、たとえたった10cmという短い距離を電気信号が伝わる速さでさえ、問題になるのです）。ということで、メモリはレジスタより格段にたくさんのことを覚えられますが、メモリを利用することは遅いのです。レジスタの何倍も遅いのです。これを覚えておくと、速いプログラムを作れるようになります。



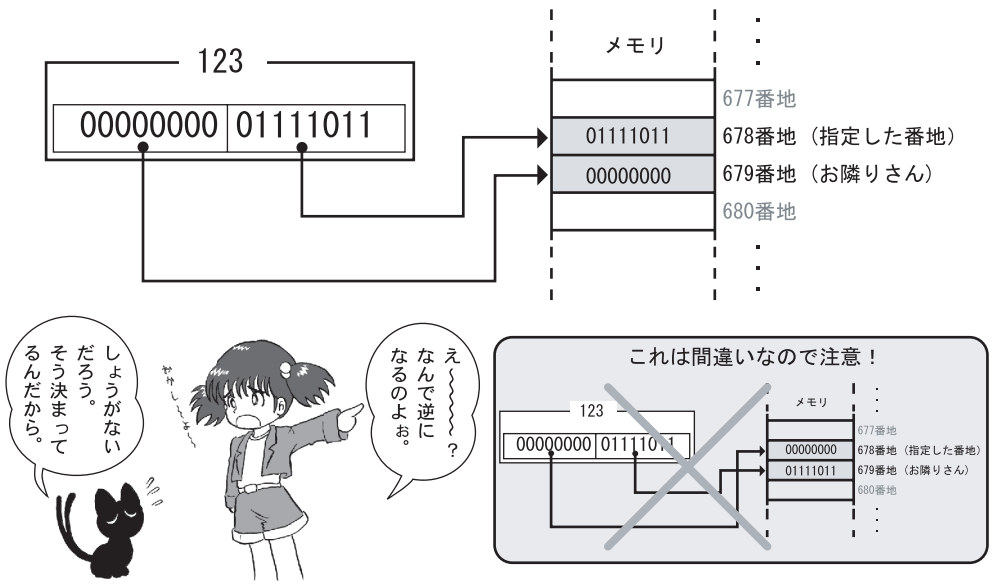
基礎知識が揃ったので、話をアセンブラに戻します。MOV命令では転送先や転送元にレジスタや定数だけではなく、メモリを指定することもできます。そしてそのときは [ ] という記号を使います。他にも、BYTE、WORD、DWORDという英語も使います。一例を示しましょう。

```
MOV BYTE [678],123
```

これは、メモリ団地の678番地に、123を覚えてもらう命令です。数字で書くとなんかまともに見えてしまうわけですが、メモリもCPUと同様に数値なんて理解していません。678というのはONとOFFが「うじゃ」っと並んだもので、その電気信号の並びが来るとちょうど回路のどこかの8個の記憶素子が反応するような回路になっていて、その8個の記憶素子が、123をONとOFFで表した電気信号を記憶するわけです。なぜ8個かというと、BYTEと指定しているからです。同様に、

```
MOV WORD [678],123
```

と書くこともできます。この場合はメモリ団地の678番地と、お隣りの679番地の記憶素子が反応します。合計16ビットです。この場合、123というのは16ビットの数値として解釈されて、つまり0000000000111011と解釈されて、まず下の01111011が678番地へ、そして上の00000000がお隣りの679番地へ送られます。



このように、アセンブラでメモリを指定するときは、

"データの大きさ [番地]"

というふうに書きます。これでひとかたまりなのです。「データの大きさ」でBYTEを指定すると指定したメモリだけが対象になります。WORDを指定するとお隣りの番地も対象になり、DWORDを指定するとお隣りとその隣りとさらにもう1つ隣りも対象になるわけです（4バイトだから）。ここで言うお隣りは、番地が増える方向のお隣りです。

メモリ番地の指定方法ですが、定数だけではなくレジスタを使うこともできます。たとえば、BYTE [SI] とか WORD [BX] とかです。SIに987が入っていれば、BYTE [SI] というのは BYTE [987] と解釈されて、987番地のメモリを指定したことになります。

番地の指定にレジスタが使えると書きましたが、使えるレジスタは限られていて、BXやBPやSIやDIだけです。残りのAXやCXやDXやSPは使えません。どうして使えないのかというと、CPUがそういう処理をするための回路を持ってないからで、もっと言うなら、そういう処理を表す機械語がないのです。ないものはできません。それだけのことです。文句がある人は、インテルのおじさんへメールしてください（笑）。筆者はインテルのおじさんへメールを書く勇気はないので、DX番地のメモリの内容をALに代入したいなー、と思ったら、

```
MOV BX,DX
MOV AL,BYTE [BX]
```

と書いて満足しております。

■■■■■

さてさて、以上をふまえると、メモリのSI番地の1バイトの内容をALに読み込め、という命令は、

```
MOV AL,BYTE [SI]
```

と書けるわけです。しかしMOV命令では、ビット数が同じもの同士でしか代入できない規則(*)があって、つまりALに代入する以上、BYTE以外はありえないので、これを省略することができちゃいます。そうすると、

```
MOV AL,[SI]
```

という表現になるわけです。おお、これでプログラム中の表現と同じになりました。長かったですねー。そんなわけで、これは、「メモリのSI番地の1バイトの内容をALに読み込め」という意味だったのです。

■■■■■

MOV命令では、ビット数が同じもの同士でしか代入できない規則：この規則にむりやり逆らって、たとえば MOV AX,CL とか書くと、アセンブラは対応する機械語が見つけられないので、翻訳エラーが出てしまう。

ADD命令は、足し算命令です。「ADD SI,1」というのは、C言語ふうに書くと、SI = SI + 1; ってことです。元になった英語は「add」で、これは加えるという意味です。

CMP命令は、比較命令です。比較命令ってなによ？　と思うかもしれませんが、ようするにif文の一部なのです。たとえばC言語では

```
if (a == 3) { あれこれ; }
```

とか書きますが、これはaを3と比較していることになります。これを機械語に翻訳するときは、まずは「CMP a,3」と書いて、どれをどれと比較するのか、それをCPUに指示しなければいけないのです。そしてその次に、等しいときにどうしてほしいとか、そういうことを書きます。

ここでは「CMP AL,0」ですので、ALを0と比較しなさい、という意味です。もとになった英語は「compare」で、これは比較するという意味です。

JE命令は、条件ジャンプ命令の1つです。条件ジャンプ命令っていうのは、比較命令の結果によってジャンプしたりしなかったりするもので、JEの場合は、比較の結果が等しければジャンプしなさい、という命令です。等しくなかった場合は、ジャンプしないでそのまま次の命令に行きます。そんなわけで、

```
CMP AL,0
JE fin
```

という2命令は

```
if (AL == 0) { goto fin; }
```

という意味になるわけです。この命令の元になった英語は「jump if equal」で、もし等しければジャンプしなさいという、まさにその通りの意味です。ちなみにequalをカタカナで書くと、イコールです。……さらについでに言うと、finというラベルは「おしまい」という意味で筆者がよく使うものです。

■■■■■

INT命令は、ソフトウェア割り込み命令というものなんですが、今ここで割り込みの仕組みを説明すると頭がこんがらがること間違いなしなので、とりあえず関数呼び出し命令の一種だと思ってください。もとになった英語は「interrupt」で、これは割り込みという意味です。

PCにはBIOSというプログラムがありまして、これはPCの基板の上にあるROM(*)という素子に組み込まれています。このBIOSというのはOS製作者がよく使いそうなプログラムをPCメーカがあらかじめ用意しておいてくれた、非常にありがたいものです。ちなみにBIOSというのは「basic input output system」の略で、直訳すると「基本的な入出力に関するシステム（プログラム）」なわけです。

ROM：読み出し専用メモリ。書き込むことはできない。しかし電源を切っても内容は消えない。「read only memory」の略。

最近のBIOSは、PCの設定画面とかも受け持っていて大忙しなのですが、とにかくこの名前の通り、もとはOS開発者のための関数の集まりだったのです。で、INT命令というのは、その関数を呼ぶために使う命令なのです。ちなみに、INTのうしろに数字を書いていますが、これをいくつにするかによって、BIOSのどの関数を呼び出すかを選ぶことができます。今回は0x10（つまり16）番の関数を呼び出しています。これは、ビデオカード制御に関する関数です。

BIOSはPCメーカが用意してくれるのですが、その使い方はあまり知られていません。でも簡単に調べられます。というわけで、筆者が書いたページの紹介です。

http://community.osdev.info/?(AT)BIOS

たとえば今回の文字表示の場合ですが、まず1文字ずつ表示しようと考えたとして、具体的にどうすればその機能の使い方が分かるかを調べてみましょう。

まず、文字表示ですから、ビデオカード関係でしょう。そうすると、INT 0x10 というのが候補に上がります。それで上記ページで、それらしいものはないかなーと探していくと、

一文字表示
- AH = 0x0e;
- AL = キャラクターコード;
- BH = 0;
- BL = カラーコード;
- 戻り値：なし
- 註：ビープ、バックスペース、CR、LFは制御コードとして認識される

というのを発見するわけです。だから、ここに書かれているようにレジスタに各種の値を代入して、それで INT 0x10 を実行すれば、めでたく1文字が出てくるというわけです(*)。

■■■■■

新出命令の最後は、HLT命令です。この命令はかなりマイナーな命令で、この命令を2日目で登場させるのは、たぶん世の中でも筆者くらいしかいないのではないかとちょっと思うわけですが、ここに筆者のこだわりがあるので、熱く語らせてください（笑）。

HLT命令はCPUを停止させてしまう命令です。しかし完全に停止するわけではなくて（完全に停止するにはもちろん電源を切らないといけません）、待機状態になります。キーボードを押すとか、マウスを動かすとか、なにかそういう外部の変化があれば、CPUは目を覚まして、プログラムの続きを実行してくれます。……で、今回のこのプログラムをよく見てほしいのですが、結局HLT命令があってもなくて

文字が出てくる：ここでBLにはカラーコードを入れることになっているので、ここを変更すれば、表示色が変わるだろうと思われる。しかし、筆者らが試したところによると、なんと変わらない。なぜ何を指定しても白になるのか詳しいことは分かっていないが、色指定が簡単にはできない画面モードなのではないかと、今のところは推測している。

も JMP fin の無限ループであることに変わりはなく、HLTなんて書かなくてもいいことなのです。だから初心者にHLT命令を紹介する人はめったにいないのです。話が長くなるだけですから。

ところが、筆者はCPUを無意味に空回しするのがきらいなのです。HLTなしで全力でJMP命令を実行させれば、当然のことながらCPU負荷は100%になって、どんどん電気を使うわけです。そんなのもったいないじゃないですか。HLTを入れるだけでCPUはほとんど「ぼけ〜」っとするようになり、電気をあまり使わなくなるのです。何もしないのに電気を使うなんてもったいないです。たとえ初心者でも、何もしないならHLTするという習慣を教えておくのはいいことなんです。いやむしろ、初心者だからこそ、今のうちからこういういいことは教えておこう、と、まあ、そういうわけです。地球にやさしく、電気代にもやさしく、ついでに、PCの寿命もほんのちょっと延びますよ、多分。

あ、それで、HLT命令の元になった英語は「halt」です。停止させる、という意味です。

■■■■■

以上、とても長い説明でしたが、これで一通り、このプログラムの説明は終わりました。まとめるとこんな感じです。

helloos.nasの一部をC言語ふうに書き直してみたもの

```
entry:
    AX = 0;
    SS = AX;
    SP = 0x7c00;
    DS = AX;
    ES = AX;
    SI = msg;
putloop:
    AL = BYTE [SI];
    SI = SI + 1;
    if (AL == 0) { goto fin; }
    AH = 0x0e;
    BX = 15;
    INT 0x10;
    goto putloop;
fin:
    HLT;
    goto fin;
```

このプログラムのおかげで、msgに書いてあるデータが1文字ずつ表示されて、データが0になったらHLTの無限ループになります。こんな仕組みで「hello, world」が表示されていたわけです。

■■■■■

おっと忘れていました。ORGの0x7c00の説明がまだでした。ORG命令の意味はさっきのでいいとして、この「0x7c00」というのは、いったいどこから出てきたものなのでしょうか。これを1234とかにしたらまずいのでしょうか？　ええまずいのです。とたんに動かなくなります。

みなさんのPCには、多分64MBとか、もしかしたら512MBくらいの、非常にたくさんのメモリが搭載されていると思います。しかしこのメモリ全体を好き勝手に使っていいのかというと、実はそうではないのです。たとえばメモリの0番地、つまり一番最初のところですが、これはBIOSくんがさまざまな用途で使っておりまして、ここを勝手に使うと、もれなくBIOSとのけんかになり、BIOSは誤動作するし、あなたのプログラムも誤動作するという、とても悲しい物語になるわけです。また一方、メモリの0xf0000番地付近は、BIOSそのものがおりまして、ここも使えません。

他にも使ってはいけないメモリ領域があちこちにありまして、OS屋はこういうことを気にしなければいけません。WindowsやLinuxを普通に使っていると、こういうめんどうなことを考えなくていいようにOSが世話を焼いてくれるわけですが、今後は私たちが世話を焼く側の人になるわけです。で、ここもダメ、あそこもダメと言っているとややこしくて分からなくなるので、地図（マップ）を作るわけです。ということでこれがメモリマップなのです。さっそくちょっと見てみましょう。

http://community.osdev.info/?(AT)memorymap

地図と言いつつ全然図になっていないわけですが、それはその、このページの著者の手抜きなのであります。ちなみにこのページの著者は、なんと筆者なのであります。みなさんごめんなさい。それで、そのページを見つめてみますと、なんかよく分からないことがいっぱい書いてありますが（それも全部筆者が悪いのです、ごめんなさいぃぃ）、しかし「ソフトウェア的用途区分」のところに、

0x00007c00 - 0x00007dff：ブートセクタが読み込まれるアドレス(*)

という見逃せない項目があるのです。今回のORGの値はまさにここの数字を使ったのです。そしてこの数字を使ったからうまくいったわけです。

ここで率直な疑問があるかもしれません。なぜ0x7c00なのか？　0x7000とかのほうが切りがよくて覚えやすいじゃないか、と。筆者もそう思うのですが、とにかく0x7c00に決まってしまったのです。決めたのは多分IBMのおじさんです。今はおじいさんになっているかもしれません。

一度決まってしまうと、これを前提に各種のOSが作られるようになったので、あとから「今回から0x7000～0x71ff番地になりましたので、みなさんよろしく」なんてことはできないのです。そういうことをすると、この新PCでは、今までのOSは改造しないと起動しませんということになって、互換性悪いねえ、とか言われて売れないのです。

これからもいろいろ「なぜ？」が出てくると思いますが、結局のところ、それはみんなIBMのおじさんたちやインテルのおじさんたちが勝手に決めたことなのです。まあ深く調べると当時なりの理由が結構あるわけですが、それを説明していると、この本は2倍の厚さになってしまうので、とりあえずかんべんしてください。

アドレス：メモリの番地のこと。address。

3 ブートセクタだけを作るように整理

今後の開発を考えると、いきなりディスクイメージ全体をnaskだけで作らせるのではなく、naskにはとりあえず512バイトのブートセクタだけを作ってもらって、残りは、ディスクイメージ管理ツールで作ってもらうほうが応用が利きます。

ということで、projects/02_day のhelloos4ができました。

まずhelloos.nasは、うしろの部分をカットしました。ブートセクタは最初の512バイトだけあればいいからです。それでこのソースはブートセクタだけのソースになったので、ソースファイル名もipl.nasに変更です。

それでasm.batを改造して、出力をipl.binに変更しました。また、ついでにリストファイルipl.lstも出力させることにしました。これは、どの命令がどの機械語に翻訳されたかを簡単に確認できるテキストファイルです。今まで出すようにしなかったのは、1,440KB分のリストファイルがあまりに巨大になるせいで、今回は512バイト分ですから問題なく出せます。

そしてmakeimg.batも作りました。これはipl.binをもとにディスクイメージファイルのhelloos.imgを作るバッチファイルです。何をしているかというと、edimg.exeという筆者作のディスクイメージ管理ツールを使って、まず空っぽのディスクイメージを読み込んで、その先頭にipl.binを書き込ませて、最後にhelloos.imgという名前で結果のディスクイメージを出力させているだけです。詳しいことはmakeimg.batそのものを見てください。

そんなわけで、コンパイルからテスト実行までの手順は、!consをダブルクリックしてから、asm → makeimg → run と3つのコマンドを入力していくことで達成されるわけです。

4 今後のためにMakefile導入

helloos4までで、とりあえず筆者がhello-osを最初に作ったときのソースプログラムと完全に同じになったわけですが、筆者はMakefileというものを使っておりましたので、それの説明をしようと思います。

Makefileというのは、かなりかしこいバッチファイルのようなものです。

■■■■■

書き方は結構単純で、まず拡張子のない「Makefile」という名前のファイルを作って、これを次の内容になるようにテキストエディタで編集します。

```
# ファイル生成規則

ipl.bin : ipl.nas Makefile
    ../z_tools/nask.exe ipl.nas ipl.bin ipl.lst

helloos.img : ipl.bin Makefile
    ../z_tools/edimg.exe   imgin:../z_tools/fdimg0at.tek ¥
        wbinimg src:ipl.bin len:512 from:0 to:0   imgout:helloos.img
```

#記号は、コメントマークです。次の「ipl.bin : ipl.nas Makefile」という記述の意味するところは、「ipl.binというファイルをもし作りたくなったら、まずipl.nasというファイルとMakefileというファイルが揃っていることを確認しなさい」ということです。そして揃っていることが確認できると、その次の行以降が自動で実行されます。

helloos.imgに関する記述もまったく同様です。途中に入っている「¥」は、1行に書ききれなくて、次の行にまたがって書いたよ、というマークです。

このMakefileを動くようにするには、make.exeを呼び出す必要があります。それでコンソールから簡単に呼び出せるように、make.batを準備しましょう。make.batはtolsetのz_new_wフォルダの中にありますので、これをコピーしてきます。

■■■■■

ここまでできたら、!consでコンソールを開いて、「make −r ipl.bin」と入力してみてください。make.exeが起動して、彼はまずMakefileファイルを読み込んで、ipl.binを作る方法を考えます。すると、ipl.binの作り方が書いてあるので、その1行が実行され、無事にipl.binはできあがります。このあとに「make −r helloos.img」と入力してみると、やはりmake.exeが起動し、作り方に沿って実行してくれます。

まあここまでは面白くも何ともないのですが、ではipl.binとhelloos.imgを一度削除しまして、再度、「make −r helloos.img」と入力してみます。すると、makeはまずすなおにhelloos.imgを作ろうとするのですが、ここで材料となるipl.binがまだ存在していないことに気がつきます。そしてipl.binの作り方がMakefileの中に書かれていることを発見して、まずipl.binを作り始めます。そして無事にエラーなくできあがったことを確認してから、helloos.imgを作ってくれます。なかなかかしこいです。

さらに今度はファイルを削除しないで、もう一度「make −r helloos.img」と入力してみますと、今度は「`helloos.img' is up to date.」というメッセージを出して、何もしません。つまり、helloos.imgはもうできあがっているから、わざわざ作り直す必要はない。と言っているわけです。ますますかしこいです。

もっと試練を与えてみましょう。ipl.nasのメッセージのところを"How are you?"に編集して保存します。その上で、ipl.binとhelloos.imgはさっきのまま残しておいて、もう一度「make −r helloos.img」

と入力してみます。するとまた作り直す必要はないと言うのかと思いきや、なんとmake.exeはもう一回ipl.binから作り直しを始めるのです。つまりmake.exeは、ファイルの存在だけではなく更新日時も見ていまして、それでファイルを作り直すべきかどうかを考えてやってくれているのです。とてもえらいです。

■■■■■

これでバッチファイルよりもかしこいことは分かったと思いますが、毎回「make −r helloos.img」と入力するのはめんどうです。ということで、ラクをするテクニックがあります。もちろん、「make −r helloos.img」という内容の、makeimg.batを作ってもいいのですが、それだとバッチファイルだらけの現状を改善できないので、違う方法です。Makefileに次の記述を書き足します。

```
#コマンド

img :
    ../z_tools/make.exe -r helloos.img
```

この記述を書き足すことで、「make img」と入れるだけで「make −r helloos.img」と同じ動作をしてくれます。さあこれでラクです。makeimg.batはもうじゃまなので消しちゃいます。ついでに、

```
asm :
    ../z_tools/make.exe -r ipl.bin

run :
    ../z_tools/make.exe img
    copy helloos.img ..¥z_tools¥qemu¥fdimage0.bin
    ../z_tools/make.exe -C ../z_tools/qemu

install :
    ../z_tools/make.exe img
    ../z_tools/imgtol.com w a: helloos.img
```

という記述も書いちゃいましょう。これで、run.batもinstall.batもいりません。いらないだけでなく、たとえば「make run」と入れるだけで、まず「make img」を実行して、それからエミュレータを起動してくれます。

今までは、すでにできあがっていた場合に何度もアセンブルをやり直させるのはムダだと思っていたので、バッチファイルを分けていました。Makefileになってからはそういうムダは起きないので、いつでも心配なく「make img」を実行させているわけです。だからいきなり「make run」しても問題なく実行できます。「make install」も同様で、とにかくディスクをドライブにセットしておけば、あとはhelloos.imgがあればそのまま、なければ全自動で作られて、そしてそれがインストールされるわけです。

■■■■■

以上をまとめて、projects/02_day のhelloos5を作りました。ついでにコマンドを増やしてありま

す。最終生成物（ここではhelloos.img）以外を削除し、ハードディスクを整理するための「make clean」、そしてソースファイル以外のすべてを削除して、さらにすっきりさせるための「make src_only」です。また、パラメータなしでmakeした場合は「make img」とみなすためのデフォルト動作も書き足しました（デフォルトは先頭に書きます）。

こんなに多機能になったのに、ファイル数は5個に減って、すっきりです。ソースファイルのように、本質的に必要なものが増えるのは悪くないのですが、バッチファイルみたいに本質的には必要ないものがうじゃうじゃといるのは気になりますからね。

これで今後はもっと快適に開発できます。

■■■■■

おおっと書き忘れていましたが、このmake.exeはGNUプロジェクトの人たちが作って、フリーソフトとして公開しているものです。GNUさんはgccの作者でもある、あのGNUさんです。ありがたや〜。

こんなペースで本当に1ヵ月後にはOSができているのだろうかと、早くも心配になってきた筆者であります。いや大丈夫、当初の予定よりは遅れているけれど、説明が多いのは最初のほうだけで、C言語中心になったら、すいすい進むはずだ。うん、そうに違いない。……と希望が多く混ざったひとりごとを言いつつ（苦笑）、またあしたー。

COLUMN-1　データも「実行」できる？　機械語も「表示」できる？

helloos5についてですが、一番最初の JMP entry を JMP msg にしたら一体どうなるでしょうか？……まず、そもそもそういうことができるのかというと、できます。naskではエラーになりません。他のアセンブラでもエラーになりません。アセンブラにおけるラベルは、結局メモリの番地を表す数値でしかないので、JMPの先にあるものが機械語なのか文字コードなのか、そんなことはおかまいなしです。

ではこれを実行したらCPUはどうなるでしょうか。まず最初の命令は、0A 0A ということになって、これは OR CL,[BP+SI] という命令を意味しますから、つまりCLレジスタとメモリのBP+SI番地の内容をOR演算（何日かあとに出てきます）して、その結果をCLにしまうのです。その先は 68 65 6C なので PUSH 0x6c65 という意味になって（これも何日かあとに出てきます）、スタックに0x6c65を記憶させます。……てな具合に、まあはっきりいってめちゃくちゃですが、しかしCPUは電気信号に従って処理していくことしかできないので、意味不明な処理をどんどん進めていきます。

その結果として、画面に変な文字が出たり、フロッピーディスクやハードディスクの大事なデー

タがいきなり上書きされたり、なんかいろいろおこるでしょう。けっしてPCが壊れたわけではないのですが（だってCPUは本当に真剣に全速力で命令の実行を続けているのですから）、しかし結果的にはPCが壊れてしまったかのような結果になります。だから、よい子のみんなは実験しないでくださいね。

しかし人間は完全ではないですから、徹夜明けにプログラムしていたら、entryと書くべきところをmsgと書き間違えてしまうことがあるかもしれません。それを完全になくすことは不可能です。そしてそのたびに大事なデータが消されたりしたら、やっていられません。ということでCPUにはこういう事故を防止するための機能があります。しかしその機能はまさにOSがいろいろ設定してあげないと有効になりません（これも何日かあとにはやりますよ）。ということで、OS開発中はこの保護機能をあてにできません。私たちOS屋は、ある程度、いつもびくびくしてOSを作っているわけです。

では逆はどうでしょうか。つまり機械語を表示させてみようということです。MOV SI,msg という記述がありますが、これを MOV SI,entry にしてみます。するとどうなるのかというと、まず文字コード B8 に相当する文字が出て（たぶん絵文字か記号か、なんかそんなものです）、そのあとは偶然にも 00 なので、つまりここで表示はおしまいです。変な動作はしません。だから実験してみてもいいです。

結局これで確認できたことは、CPUもメモリも、自分が扱っている電気信号が何を意味するものなのかなんて、まったく考えてないってことです。だからデジタルカメラで風景の写真を撮って、それをディスクイメージとしてディスクに保存したら、なんと世界でもっとも優秀なOSとして動作した！　ということも、ひょっとしたらあるかもしれないのです。まあ常識的に考えれば、いろいろ誤動作するだけだとは思いますが。逆に、適当に作った実行ファイルを画像として見てみたら、世界最高の芸術作品だった、ということもありえないことではないのです。まあたいていは、フォーマットが違いますと言われるか、もしくはぐちゃぐちゃな絵が出るだけだろうと思いますが。

Chapter 3

三日目

32ビットモード突入とC言語導入

- さあ本当のIPLを作ろう
- エラーになったらやり直そう
- 18セクタまで読んでみる
- 10シリンダ分を読み込んでみる
- OS本体を書き始めてみる
- ブートセクタからOS本体を実行させてみる
- OS本体の動作を確認してみる
- 32ビットモードへの準備
- ついにC言語導入へ
- とにかくHLTしたい（harib00j）

1 さあ本当のIPLを作ろう

昨日までのブートセクタは、IPL（初期プログラムローダ）と言いつつも、実はちっともプログラムなんてロードしていなかったわけで、でも今日からは本当にロードすることにします。

さて意気込みはこれでいいとして、今後はこのOSをhello-osなんていう名前で呼ぶのはつまらないので、名前を変えることにしました。その名も「はりぼてOS」です。はりぼてっていうのは、映画の撮影のときに出てくる岩とかが、実はにせもので中身は空洞（くうどう）みたいな、そういうもののことです。つまりこのOSは見せかけはOSだけど、中身はからっぽでいいんだ、無理することはないんだ、気楽にいこうよ、っていう気持ちです。

今後はずっと名前が「はりぼてOS」ですし、しばらくの間はOSではなく一種のデモプログラムとして作っていくわけですが、しかしそれでも最後にはちゃんとOSになりますので、心配しないでください。

ところで、いきなり話は脱線しますが、よく考えてみれば、はりぼてなのはOSだけじゃないんですよね。そもそもCPUだって、本当は数の概念なんかちっとも理解していないのに、0011と0110という電気信号を同時に送ったら1001という電気信号が出てくるような回路をちょっと作っただけで、これが加算回路

だとかいうことになるわけですよ。これが3+6=9のことだって思っているのは人間だけで、CPUはただ電気信号をピコピコやっているだけですよ。つまり数なんてちっとも分からなくても、分かったふりして計算結果を出すことはできるわけで、これだってはりぼてかなと思うのです。
ゲームプログラムを作ったことがあれば分かると思うのですが、たとえばコンピュータ対戦の将棋をやったとして、コンピュータはなかなかうまい手を指してくるわけですが、しかしあれだってコンピュータは将棋のルールなんてこれっぽちも分かってないんです。ただプログラム通りにやっているだけなんですから。コンピュータがすごく厳しい手を指してきたとしても、それは別に容赦がないとか、頭がいいとか、プレイヤのことを嫌っていて思いやりがないとか、勝ちたくてしょうがなくて、勝利への執念がそうさせているのだとかいうことはまったくなくて、まるでそうであるかのようにやっているだけなんです。つまり中身は空っぽ、だからはりぼて。はりぼてばんざい！　……だからOSがはりぼてであって何が悪いんだ。いや悪くない。これでいいのだー。

■■■■■

さあ、まずは簡単なプログラムからいきましょう。ディスクの最初の512バイトはブートセクタなので、その次の512バイトを読み込んでみます。ということで、さっそくプログラムを改造しました。projects/03_day のharib00aです。いつものように、tolsetの中にコピーしてください。

さて、書き足された部分はおおざっぱに言うと、これだけです。

今回書き足された部分

```
MOV     AX,0x0820
MOV     ES,AX
MOV     CH,0            ; シリンダ0
MOV     DH,0            ; ヘッド0
MOV     CL,2            ; セクタ2

MOV     AH,0x02         ; AH=0x02 : ディスク読み込み
MOV     AL,1            ; 1セクタ
MOV     BX,0
MOV     DL,0x00         ; Aドライブ
INT     0x13            ; ディスクBIOS呼び出し
JC      error
```

新出命令はなんとJCだけ。いいですねえ、こういうのって。説明がラクですよ。JCというのは「jump if carry」の略で、つまり、キャリーフラグが1だったらジャンプしなさいというそれだけのことです。キャリーフラグっていきなりなんだよ、と思っても心配はいりません。すぐに分かります。

■■■■■

とにかく命令の意味は分かるけど、何が起きるのか分からないのは「INT 0x13」だと思うので、調べてみます。もちろん毎度の(AT)BIOSで。

http://community.osdev.info/?(AT)BIOS

すると、

・ディスクからの読み込み、ディスクへの書き込み、セクタのベリファイ、およびシーク
　・AH = 0x02;（読み込み時）
　・AH = 0x03;（書き込み時）
　・AH = 0x04;（ベリファイ時）
　・AH = 0x0c;（シーク時）
　・AL = 処理するセクタ数;（連続したセクタを処理できる）
　・CH = シリンダ番号 & 0xff;
　・CL = セクタ番号(bit0-5) | (シリンダ番号 & 0x300) >> 2;
　・DH = ヘッド番号;
　・DL = ドライブ番号;
　・ES:BX = バッファアドレス;（ベリファイ時、シーク時にはこれは参照しない）
　・戻り値：
　・FLAGS.CF == 0 ： エラーなし、AH == 0
　・FLAGS.CF == 1 ： エラーあり、AHにエラーコード（リセットファンクションと同じ）

というのを見つけることができます。この場合、AHは0x02なので、おお、読み込みですね。

■■■■■

それで「戻り値」の欄を見るとFLAGS.CFというよく分からない記述がありますが、これはキャリーフラグのことです。つまりこの関数を呼び出すと、エラーがない場合はキャリーフラグが0、エラーがある場合はキャリーフラグが1になるわけです。それでさっきのJC命令の存在理由が分かるわけです。

キャリーフラグというのは、1ビットしか記憶できないレジスタで、CPUには他にもいくつかこのような1ビットしか記憶できないレジスタを持っています。こういうレジスタのことをフラグと言っています。フラグっていうのはflag、つまり旗（はた）ですね。旗の上げ下げで状態を表すのにたとえられて、この名前になったようです。

キャリーフラグというのは、本来はキャリー（carry）という状態を表すために使うものなのですが、CPUのフラグの中ではキャリーフラグが一番扱いやすいので他の用途にもよく使います。今回もエラーの有無の報告に使われているわけです。

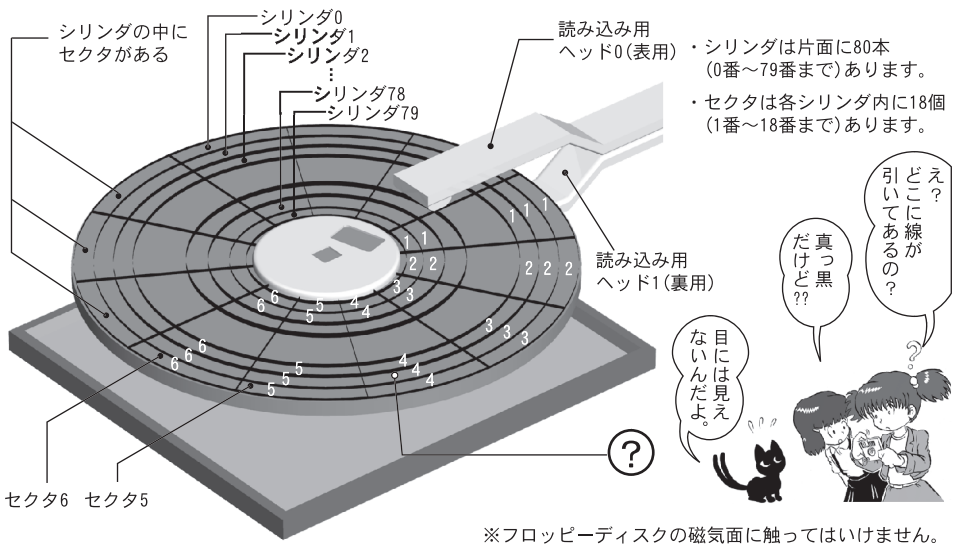
■■■■■

残りを確認していくとしましょう。CH, CL, DH, DLにはそれぞれ、シリンダ番号、セクタ番号、ヘッド番号、ドライブ番号というものを代入しなければいけません。今回のプログラムでは、シリンダ番号0、ヘッド番号0、セクタ番号2、ドライブ番号0、となっています。

ドライブ番号というのは、フロッピーディスクドライブがたくさんつながっていたときに、どのドライブのディスクから読み込むのかを指定するためのものでした。今のPCは、フロッピーディスクドライブが1つのものばかりですが、昔は2つくらいはあったのです。とにかく今は1つしかないので、迷うことなく0番を指定します。

これでどのドライブから読むかが決まりました。次はそのディスクのどこから読むか、です。

もしいらないディスクを持っていたら、それをバキっと割ってみてほしいのですが、中には8cmのCDを真っ黒にしたような円盤が入っています。円盤というか、ぺらぺらの磁気フィルムですが。これは外側から、シリンダ0、シリンダ1、……、シリンダ79という輪が集まったものでありまして、いやもちろん工場でそのように生産しているという意味ではなく、データを記録する装置からすると、とりあえずそういうふうに考えているというだけなんですが、だからとにかくここでもそういう説明をします。で、だからディスクには全部でシリンダが80個あるんです。ちなみにシリンダのスペルは、cylinderで、円筒のことです。きわめて背の低い円筒が、同心円状に重なったような状態を思い浮かべて命名したのでしょうか。



※フロッピーディスクの磁気面に触ってはいけません。

お次はヘッドです。ここで言うヘッドというのは磁気ヘッドのことで、つまり表から磁気ヘッドを当てるか、裏から磁気ヘッドを当てるか、ということです。フロッピーディスクはCD-ROMとは違って両面記録方式なんです。ヘッド0とヘッド1があります。

最後にセクタです。シリンダとヘッドを指定したら、その円周に沿って磁気を何ビットも記録できるわけですが、そのままではちょっと量が多すぎるので、さらにいくつかに等分します。フロッピーディスクでは18等分されています。こうして分けられた1つがセクタです。1つの円周には18セクタあって、これは、セクタ1、セクタ2、……、セクタ18と呼ばれています。セクタのスペルはsectorで、領域とか扇形という意味です。

以上をまとめますと、1枚のディスクには80シリンダあって、ヘッドは合計2つあって、さらにそれぞれ18セクタあって、しかも1セクタは512バイトなので、

80×2×18×512＝1,474,560バイト＝1,440KB

となっているわけです。IPLが入っているブートセクタはC0-H0-S1だったわけですが（これは、シリンダ0、ヘッド0、セクタ1の略のつもり）、この次のセクタはC0-H0-S2なのです。それでこれを指定してみたというわけです。

■■■■■

あと分からないのはバッファアドレスというやつですね。これはつまり、メモリのどこに読み込むか、それを表す番地のことです。普通、番地は1つのレジスタで表せばよさそうなものですが、BXだけにしてしまうと0～0xffffまでの値しか表せず、これはつまり、0から65,535番地までしか表せないということで、みなさんのPCのメモリは64MBとか、もしかしたらもっと多いかもしれないのに、なんとたったの64KBまでしか使えないわけです。これは悲しいです。

これを克服するために、EBXレジスタというのがあとになってできて、これで4GBまで扱えるようになりました。これはCPUが扱える最大メモリ容量なので、問題ありません。しかし、EBXが使えるようになったのはもっとずっとあとの話で、BIOSくんたちが設計された時代のCPUは、32ビットレジスタをつけるなんてちょっとできなかったらしく、しょうがないので、補助的な役目をするセグメントレジスタというものを作りました。そしてメモリの番地を指定するときに、このセグメントレジスタも使えるようにしたのです。

ES:BXという表現はまさにそれで、MOV AL,[ES:BX] などと書いて使います。この場合のメモリの番地は、ES×16＋BXで計算することになっています。イメージとしては、ESレジスタで番地をおおざっぱに決めておいて、BXで細かく指定、みたいな感じです。

これでESに0xffff、BXにも0xffffを入れることで、1,114,095バイト、つまり約1MBまでメモリの番地を指定できるようになったのです。これでも64MBにはまったく届いていないのですが、当時の

インテルのおじさんは、これでいいやと思ってしまったようです。元祖BIOSくんが設計されたのも、ちょうどこの仕組みで満足していた時代のことだったので、私たちも今回は当時のルールで使わないといけないのです。だからとりあえず、1MBまででガマンすることにしましょう。

今回は、ES＝0x0820 、BX＝0 なので、このディスクのデータが読み込まれるのは0x8200番地から0x83ff番地までになります。なんか切りが悪いな、0x8000からでいいじゃないかと思うでしょう。でも0x8000～0x81ffの512バイトにはあとでブートセクタの内容を入れようかなと思ったのです。だからこれでいいかなーって。

なんで0x8000以降を使うことにしたのかというと、特に深い理由はないのですが、メモリマップを見たらこのへんは誰も使っていないようだったので、「はりぼてOS」が使わせてもらうことにしました。0x7c00～0x7dffはブートセクタが使っていますが、0x7e00以降は誰も使っていないようで、0x9fbffまではOSが好きに使っていいのです。

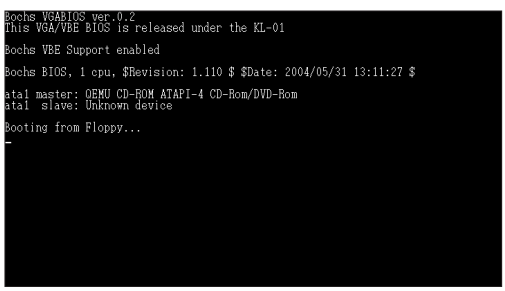
■■■■■

今までセグメントレジスタなんて全然考えないでやってきましたが、実はどんなメモリの番地指定でも、セグメントレジスタをいっしょに指定しなければいけない、という決まりがあります。省略するとほとんどの場合でDS:を指定したものとみなされます。

今まで私たちが MOV CX,[1234] と思っていたのは、MOV CX,[DS:1234] という意味だったのです。MOV AL,[SI] も MOV AL,[DS:SI] という意味なのです。アセンブラでは毎回書くのはめんどうなので、省略してもいいことになっているんです。

こんなルールがあったので、DSを0にしておかなければいけなかったのです。もしDSが0でなければ、その16倍の値が番地にいつも足されることになるので、どこか変なところを読み書きしてしまったかもしれませんから。

さてこのプログラムを実行してみると、えーと、エラーが起きたらエラーメッセージが出ますが、まあエラーは起きないでしょう。そうなると、何もしません（笑）。だから何も出なかったら大成功です。



大成功～

ああそういえば書き忘れましたが、Makefileではちょっとした変数が使えるので、それを使って書き直しています。どうです？　前よりもちょっと読みやすくありませんか？

2 エラーになったらやり直そう

フロッピーディスクというのは結構いいかげんなものなので、たまにうまく読めないこともあります。そういうときは気を取り直してもう一回読めばいいのです。ということで、エラーが一回出たくらいであきらめないで、やり直しをさせることにします。ああでも、永遠に何度もやり直していると、本当にディスクの寿命を迎えてしまったときに永久ループになってしまって終われなくなるので、5回くらいで本当にあきらめることにします。その改良をしたのが projects/03_day のharib00bです。

今回書き足した部分など

```
; ディスクを読む

        MOV     AX,0x0820
        MOV     ES,AX
        MOV     CH,0            ; シリンダ0
        MOV     DH,0            ; ヘッド0
        MOV     CL,2            ; セクタ2

        MOV     SI,0            ; 失敗回数を数えるレジスタ
retry:
        MOV     AH,0x02         ; AH=0x02 : ディスク読み込み
        MOV     AL,1            ; 1セクタ
        MOV     BX,0
        MOV     DL,0x00         ; Aドライブ
        INT     0x13            ; ディスクBIOS呼び出し
        JNC     fin             ; エラーがおきなければfinへ
        ADD     SI,1            ; SIに1を足す
        CMP     SI,5            ; SIを5と比較
        JAE     error           ; SI >= 5 だったらerrorへ
        MOV     AH,0x00
        MOV     DL,0x00         ; Aドライブ
        INT     0x13            ; ドライブのリセット
        JMP     retry
```

まずは新出命令から。JNCというのは条件ジャンプ命令の1つで、「jump if not carry」の略です。つまりキャリーフラグが0ならジャンプするんです。またJAEもやっぱり条件ジャンプで、「jump if above or equal」の略です。日本語にすると、大きいかもしくは等しければジャンプしなさい、ということです。

エラーがあった場合の処理ですが、読み込み直す前に、AH＝0x00、DL＝0x00で、INT 0x13 しています。これは例の(AT)BIOSのページによると、「システムのリセット」です。これでドライブをリセットしてしゃんとさせて、それでやり直します。あとはプログラムを読めば分かるでしょう。

うーん、今日はすいすい進むなあ。どんどんいこう。

3 18セクタまで読んでみる

気分が乗ってきたので、調子に乗ってどんどん先のセクタを読んでしまおうではないですか。ということで、projects/03_day のharib00cです。

今回書き足した部分など

```
; ディスクを読む

        MOV     AX,0x0820
        MOV     ES,AX
        MOV     CH,0            ; シリンダ0
        MOV     DH,0            ; ヘッド0
        MOV     CL,2            ; セクタ2
readloop:
        MOV     SI,0            ; 失敗回数を数えるレジスタ
retry:
        MOV     AH,0x02         ; AH=0x02 : ディスク読み込み
        MOV     AL,1            ; 1セクタ
        MOV     BX,0
        MOV     DL,0x00         ; Aドライブ
        INT     0x13            ; ディスクBIOS呼び出し
        JNC     next            ; エラーがおきなければnextへ
        ADD     SI,1            ; SIに1を足す
        CMP     SI,5            ; SIを5と比較
        JAE     error           ; SI >= 5 だったらerrorへ
        MOV     AH,0x00
        MOV     DL,0x00         ; Aドライブ
        INT     0x13            ; ドライブのリセット
        JMP     retry
next:
        MOV     AX,ES           ; アドレスを0x200進める
        ADD     AX,0x0020
        MOV     ES,AX           ; ADD ES,0x020 という命令がないのでこうしている
        ADD     CL,1            ; CLに1を足す
        CMP     CL,18           ; CLと18を比較
        JBE     readloop        ; CL <= 18 だったらreadloopへ
```

新出命令はJBEですね。これも条件ジャンプ命令です。「jump if below or equal」の略です。日本語にすると、小さいかもしくは等しければジャンプしなさい、ということです。

やっていることは非常に単純で、プログラムを読めばすぐに分かるでしょう。次のセクタを読むために必要なことは、CLを1増やすことと、ESを0x20だけ増やすことです。CLはセクタ番号ですし、ESは読み込み番地の指定のためですからね。0x20というのは、512を16で割った値を16進数で書いたものです。すなおに、ADD AX,512/16 って書いたほうが分かりやすかったかもしれませんね（筆者は自然に頭の中で暗算してしまって0x20と書いてしまいましたが、もちろん512/16と書いても同じになります）。ああ、BXに512を足すほうがラクじゃないかって？　言われてみればその通りでした。でも今回はESに足し算する練習をやりたかったので、とりあえずここはこのままで。

ところで、なんでこんなものをループにするんだと思った人はいますでしょうか？　あなたはするどい。そう、確かにこれはループにする必要はなくて、ディスク読み込みの INT 0x13 のところで、ただ

ALを17にしておけばすむのです。そうすれば、セクタ2～18の、合計17セクタが無事に問題なく読み込まれますから。これをループにしたのは、ディスクBIOSの読み込みファンクション(*)の説明のところの「補足」のところを気にしたからです。書き抜きますと、

・処理するセクタ数は0x01～0xffの範囲で指定（0x02以上を指定するときは、連続処理できる条件があるかもしれないので注意 -- FDの場合は、たぶん、複数のトラックにはまたがれないし、64KB境界をこえてもいけない、だと思います）

と、なんかややこしそうなことが書いてあるわけです。めんどうなので詳細は省きますが、結論から言うと、この注意書きは今のところは該当しないので、やっぱり AL=17 でも同一の動作になります。しかし次のプログラムではこれが問題になるので、今後の発展を考えて、あえて1セクタずつのループにしました。

相変わらず画面表示上の変化はないままですが、これでC0-H0-S2からC0-H0-S18までの512×17＝8,704 バイトが、メモリ番地0x8200～0xa3ffに読み込まれたわけです。

4 10シリンダ分を読み込んでみる

調子がいいのでがんがんいきます。C0-H0-S18のセクタの次は、ディスクの裏側へ回ってC0-H1-S1です。これも0xa400から読み込んでしまいましょう。さらに、このままC0-H1-S18まで読み進んだあとは次のシリンダへ行って、C1-H0-S1を読みます。この調子でどんどん読み進んで、C9-H1-S18まで読み込ませてみたいと思います。ということで、projects/03_day のharib00dです。

今回書き足した部分など

```
; ディスクを読む

        MOV     AX,0x0820
        MOV     ES,AX
        MOV     CH,0            ; シリンダ0
        MOV     DH,0            ; ヘッド0
        MOV     CL,2            ; セクタ2
readloop:
        MOV     SI,0            ; 失敗回数を数えるレジスタ
retry:
        MOV     AH,0x02         ; AH=0x02 : ディスク読み込み
        MOV     AL,1            ; 1セクタ
        MOV     BX,0
        MOV     DL,0x00         ; Aドライブ
        INT     0x13            ; ディスクBIOS呼び出し
        JNC     next            ; エラーがおきなければnextへ
        ADD     SI,1            ; SIに1を足す
        CMP     SI,5            ; SIと5を比較
        JAE     error           ; SI >= 5 だったらerrorへ
        MOV     AH,0x00
        MOV     DL,0x00         ; Aドライブ
        INT     0x13            ; ドライブのリセット
        JMP     retry
```

ファンクション：機能とか関数という意味。function。

試し読みはお楽しみ
いただけましたか？

ここからはManatee
おすすめの商品を
ご紹介します。

Tech Book Zone Manatee

3.15
2017

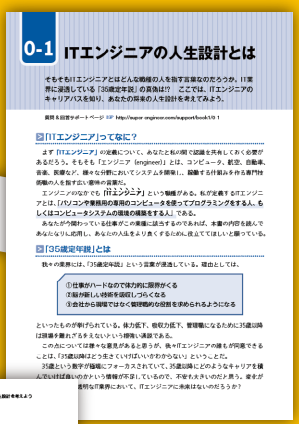
おすすめ電子書籍

Tech Book Zone Manatee

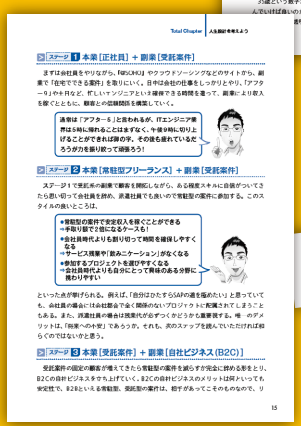
1

副業? 独立? それとも?
ITエンジニアの人生設計の決定版!

組織を束ねるマネジャーになるか、現場のスペシャリストであり続けるか。自分の技術を活かして独立するか、副業を考えるか……。意外に悩ましいITエンジニアの人生設計。会社に依存しない、転職や独立も射程に入れた「マインドセット」の持ち方から、お金、営業戦略、顧客対応術、ビジネスモデルの構築といった「ビジネスロジック」まで。求人情報ポータルサイト「@SOHO」の開発者が、自身の知見と経験から得たノウハウを教えます。



トピックスごとにオンラインの公式サイトで質問コーナーを用意。疑問に答えてくれる



ITエンジニアがどのように人生を描き切り開いていけるか、著者の実体験をもとに解説

「開発技法」

推奨職種を併記。アーティストやレベルデザイナなど専門ごとに読むべき章がわかる

プログラムやスクリプトの勉強をしたことがない人でも読み進められる

2

UE4におけるゲーム制作で
必須の知識と経験が身につく!

『Unreal Engine 4で極めるゲーム開発』は、Unreal Engine 4（UE4）の機能を単に紹介するだけでなく、一本のサンプルゲーム開発に順序よく盛り付けていく構成になっており、機能の組み合わせ方や、実践的な使い方をラーニングできます。3Dゲーム開発の一般的なワークフローやプロセスも解説し、章の構成も実際の開発プロセスに近づけました。今使っている人にも、これから始める人にも、すべてのUE4ユーザーにオススメの一冊!

開発系読み物

IT エンジニアのための「人生戦略」の教科書
技術を武器に、充実した人生を送るための「ビジネス」と「マインドセット」

マイナビ出版
平城寿（著者） 256 ページ
価格：2,462 円（PDF・EPUB）

開発ツール

Unreal Engine 4 で極めるゲーム開発

ボーンデジタル
湊和久（著者）
592 ページ
価格：4,860 円（PDF）

Tech Book Zone Manatee

「開発技法」

3.15 2017

ワークフローを疑似体験！ GitHub が初歩からわかる

開発ツール

3

Web 制作者のための GitHub の教科書 チームの効率を最大化する 共同開発ツール

Web 制作における「GitHub」の使い方が、実際のワークフローをイメージしながら理解できます。「そもそもどんなサービスなの ?」「どういうときにどの機能を使えばいいの ?」といった初歩の疑問から解説します。

インプレス
塩谷啓・紫竹佑騎・原一成・平木聡（著者）
224 ページ　価格：2,052 円（PDF）

Docker が利用される 現場のノウハウが凝縮！

開発ツール

4

Docker 実践ガイド

Docker が利用される環境や背景をはじめ、導入前のシステム設計、基本的な利用方法、Dockerfile による自動化の手法、プロセッサ、ネットワーク、ストレージなどの資源管理、管理・監視ツールについて解説します。

インプレス
古賀政純（著者）
328 ページ　価格：3,240 円（PDF）

チーム改善に活かす ITIL 悩めるリーダーにオススメ

開発系読み物

5

新米主任 ITIL 使ってチーム改善します！

化粧品メーカーで主任に昇格した友原京子。異動先は問題だらけのハチャメチャ部署だった…。『新人ガール ITIL 使って業務プロセス改善します !』の第 2 弾。英国生まれの IT 運用ノウハウ「ITIL」をチーム改善に活かします。

シーアンドアール研究所
沢渡あまね（著者）
304 ページ　価格：1,750 円（PDF）

プロトタイピングによって 初期段階での可能性を探る

開発系読み物

6

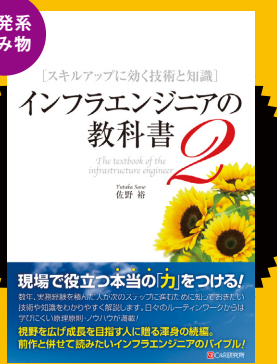
プロトタイピング実践ガイド スマホアプリの効率的なデザイン手法

本書で解説するプロトタイピングは、紙などを使った「低精度プロトタイピング」を中心とした手法です。設計フェーズの早期段階から作成し、検証と改善によって、機能要件や UI 設計、デザインを具現化していけます。

インプレス
深津貴之・荻野博章（著者）
240 ページ　価格：2,592 円（PDF）

インフラエンジニアの 必須知識をていねいに解説

開発系読み物



7

インフラエンジニアの教科書 2 スキルアップに効く技術と知識

数年間インフラエンジニアの経験を積んでいても「自分は詳しく知らないし、他の人に説明できない」といったことがあります。本書は実務経験を積んだインフラエンジニアを対象に、必須知識をわかりやすく解説します。

シーアンドアール研究所
佐野裕（著者）　価格：2,070 円（PDF・EPUB）

エミュレータ制作を通して コンピュータの中身を理解

開発系読み物

8

自作エミュレータで学ぶ x86 アーキテクチャ コンピュータが動く仕組みを徹底理解！

機械語やアセンブリ言語が CPU でどう実行されるか意識することはめったにありません。本書ではエミュレータの制作を通して x86 CPU の仕組み、メモリ・キーボード・ディスプレイといった部品と CPU の関わりを学びます。

マイナビ出版　内田公太・上川大介（著者）　196 ページ
価格：2,324 円（PDF）